Porting the Lattice Boltzmann code to OpenCL

Name: *Ioannis Borektsioglou (ib16118),* ***Candidate No.:*** *33463*

Introduction

The aim of this assignment is to take a serial implementation of Lattice-Boltzman method and make it run as fast as possible with OpenCL running on Nvidia Tesla K20 GPUs in BlueCrystal Phase 3. The main issues of this problem are first, how to port all main functions to OpenCL kernels which can be merged and second, how we can deal with overheads when we try to calculate the average velocity.

This report describes the way that we port all the main functions to OpenCL kernels by transferring the data from CPU to GPU and backwards in a sensible way and how we access data to memory. It also describes what the overheads of average velocity reduction are and how we can optimise them. Apart from that, it talks about some minor optimisations concerning OpenCL and finally makes comparisons among deferent sizes of work groups and draws conclusions about how fast OpenCL is comparing to OpenMP and MPI.

OpenCL Design

OpenCL is a library which runs generally on GPU. In order to make the code run in GPU we have to port the main functions into kernels and pass the data (cells, tmp_cells, obstacles) into our device, which is GPU. The code, we have been given, passes data from host to device in each timestep which is very expensive and unnecessary. It is very expensive because reading and writing take much time to finish and this occurs to each timestep. We choose to keep the data into device and transfer them to host when all timesteps finish, outside the loop. We port all main functions to kernels but from previous assignments, propagate rebound and collision are already merged. We keep cells, tmp_cells and obstacles as global memories because we do not need to transfer data through the work-items. The only step where we need to share data is propagate. We use private memory to do this and we take the values from adjacent cells. Then we read from that private memory to tmp_cell and finally we swap the pointers in host outside the function because the values that we need are in tmp_cells. This swap is executed in the memory which is stored inside the device. Finally for the calculation of the average velocity, we pass a local memory to kernels with size equal to the size of work groups and a global memory with size equal to number of work groups. This global memory is created inside the initialise function and it passes to device only one time outside the loop. We calculate velocity for each work-item and we store it into the local memory.

We have a simple reduction function [1] where we pass the two memories and only one thread of each work group sum the values corresponded to this specific work group. In host we read back the global memory which contains the average velocity for each workgroup and we add all the values. With these steps and procedures we have a speed up for 256x256 equal to 18.5. Starting with a serial time for this data file equal to 210.99s we end up to 11.40s (Table 1).

	Serial Time	**OpenCL Time**	**SpeedUp**
128x128	24.1s	3.00s	8x
128x256	51.19s	3.97s	12.9x
256x256	210.99s	11.40s	18.5x
1024x1024	831.95s	29.53	28.1x

Table 1: Serial vs naive OpenCL timing

OpenCL Optimisation

The time that this implementation was achieved is quite low but there are several optimisations that could be done in order to run the code faster. First of all, the way that the grid is stored is Array of Structure (AoS). Although AoS is natural to code, it is time consuming. This happens because adjacent work-items like to access adjacent memory locations. We transform our data on the host so as to have coalesced memory accesses, which are highly important in high performance computing. Structure of Arrays (SoA) suits memory coalescence in vector units. Therefore, we construct a matrix which is 9 times bigger than the initial one and it contains float numbers instead of structures. Each number represents each speed of each tile. We store the same speeds together so when thread i accesses memory location n for a speed value, thread i+1 will access memory location n+1 for the same speed. This approach has a significant effect; for example in 256x256 dataset we achieve a speedup of 1.6x. Table 2 shows the speedup with this optimisation for all datasets.

	Arrays of Structure	Structure of Arrays	SpeedUp
128x128	3.00s	2.25s	1.3x
128x256	3.97s	2.75s	1.4x
256x256	11.40s	7.03s	1.6x
1024x1024	29.53s	14.64s	2x

Table 2: SpeedUp with SoA

Another significant optimisation is the calculation of average velocity. The reduction that was used is simple but slow, which occurs because of two reasons. The first is that only one thread of each work group adds the values and the second is that we transfer the data to the host in every timestep. There are different optimisations in order to avoid the fact that only one thread makes the operation [2]. We use a tree-based approach for each work group which has complexity O(dlogn), where d is the number of work groups and n is the size of each work group. For example if we have a vector of 4 numbers [a,b,c,d], we need to compute a+b+c+d = (a+b)+(c+d). (a+b) can be computed in parallel with (c+d), and then the partial reduction can be combined to complete the reduction. This can be generalised to arbitrary sizes. Figure 1 illustrates the reduction tree on eight element vector and how this is mapped onto a SIMD processor. The runtime of 256x256 dataset was reduced from 7.03s to 6.79s which is a speedup equal to 1.03. In Figure 1 we can also see how each work-item is used during this reduction. The active work-items get sparser and sparser at each step which leads to poor SIMD efficiency. The addition operator is both associative and commutative. We try to take advantage of this to introduce an improved version of reduction. We can use commutativity to reorder the operations into a more SIMD friendly structure as shown in Figure 2. The active work-items are compacted into contiguous blocks with a commutative reduction tree. This reduction leads to better SIMD efficiency; in fact the runtime of 256x256 dataset was reduced to 6.63s.

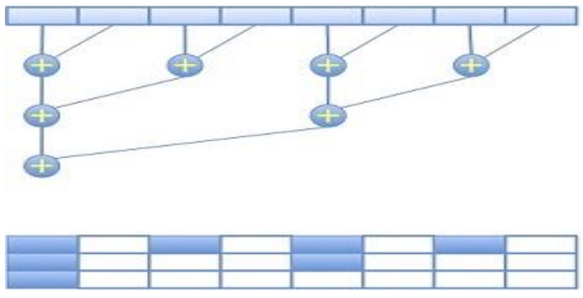
Figure 1: Associative reduction tree and SIMD mapping

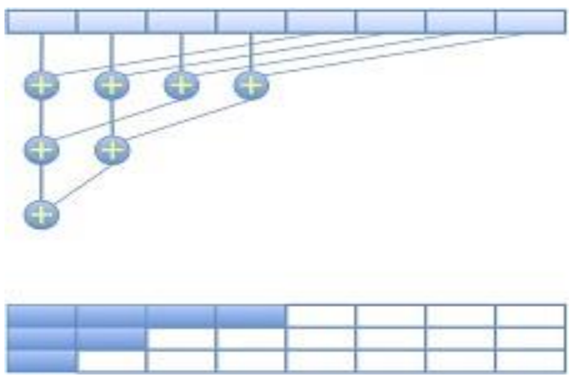
Figure 2: Commutative reduction tree and SIMD mapping

However, we have another reason that reduction is slow. When we compute the sums of each work-group we read them from device at each timestep. We construct a bigger matrix with size maxIters*workgroup_size where we store the sums for each workgroup at each iteration and when the whole loop finishes we read this matrix from

	Time	Step Speedup	Cummulative Speedup
Initial Reduction	7.03s		
Associative Reduction	6.79s	1.03x	1.03x
Commutative Reduction	6.63s	1.02x	1.06x
Read once	5.59s	1.18x	1.25x

Table 3: Performance of reduce optimisations

the device to host and we add all the values to compute the average velocity. Table 3 shows the speedup for each optimisation of reduction for 256x256 dataset. At this point local memory for tmp_cells was used, but the execution time was increased from 5.59s to 5.63s. This happened because local memory is used instead of global to share data between the work-items. The only step where we move data through work-items is the propagate step where we use private memory which is faster. Therefore, the execution time is approximately the same. Finally, in host, we set the arguments of the functions, which remain unchanged, outside the loop. Those which change, such as cells, tmp_cells and the index of the iteration (it is used in reduction), are set continuously within every timestep. With the latter optimisation the execution time was reduced to 4.3s for the 256x256 dataset.

Results and Comparisons

A comparison between the block sizes was carried out in order to decide which is the optimal. Figure 3 shows the graph between the runtime and the size of only one dimension of block for 256x256. As we can see there is an optimal value around the middle. This happened because from the one hand we increase block size (when block size is high) and from the other hand we increase the number of blocks (when block size is low). We also see that the value 8x8 = 64 ends up to a time around 4 seconds whereas 16x16 =256 ends up to an execution time 0.04s higher. More specifically the final execution time with block size 8x8 = 64 for 256x256 dataset averaged over 5 times is 4.29s.

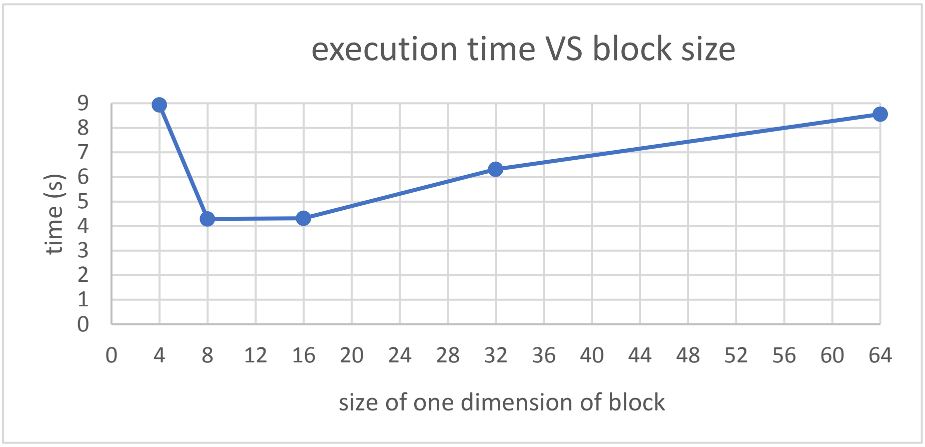

Figure 3: Compare execution time vs block size

With OpenMP the same serial code, which runs at 210.99s, was reduced to 13.5s (15.62x speedup) for 256x256 dataset and with MPI was reduced to 5.63s (37.47x speedup). With OpenCL, we achieved a speedup equal to 49.18x, which happens because first GPU has twice as much bandwidth as CPU and then we have more threads in GPU comparing to CPU.

References

[1] https://handsonopencl.github.io/

[2] http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/