Lattice Boltzman simulation in OpenCL

José Hernández del Pino (JH3875)
High Performance Computing Student
University of Bristol
United Kingdom
jose.hernandez.2011@my.bristol.ac.uk

***Abstract*—this document describes the efforts made to convert a reference serial C implementation of a Lattice Boltzmann simulation to OpenCL.**

Keywords-OpenCL; Boltzmann; GPU

I. INTRODUCTION

In this report we will present the various techniques used to port a reference serial C implementation of the Lattice Boltzmann scheme over to OpenCL. Because some of the serial code optimisations from previous exercises do not lend themselves to easy migration to OpenCL, we have chosen to start from the original non-optimised serial code and apply optimisations that are compatible with OpenCL on a case by case basis. Briefly:

- The source code has been restructured to maximise the number of potentially parallelisable regions (Chapman, Jost and van der Pas 2007). Namely, the acceleration, collision and average velocity calculation functions have been reordered to allow all resulting kernels to be combined to limit the cost associated with launching a larger number of different kernels. This also speeds up our program by reducing memory access overheads since we are now able to reuse of data which is already in private or local memory.
- Cells can read but do not modify neighbouring cell values to avoid unnecessary synchronisation (Bella, et al. 2002).
- Input and output buffers are switched intermittently between time steps using just pointers to minimise memory copying and eliminate the need for temporary buffers. This also removes the need for expensive global memory synchronisation which would otherwise had to be enforced by splitting up kernels.
- Data structures have been reworked to allow coalesced access to global memory by all threads in each workgroup (David and Hwu 2010).
- The numbers of workitems, workgroups and workload per workitem have all been calculated to ensure maximum occupancy. E.g. the number of workgroups is a multiple of the number of compute units on the GPU.
- Use integer data types instead of chars to take advantage of preferred data alignments on the target architecture.

All these points are discussed in more detail in later sections of this document.

II. FROM HOST TO DEVICE CODE

Our strategy from moving host code to the OpenCL device has been to move each function in turn, keeping both the original host code and the OpenCL device code running simultaneously so that we could compare the results and ensure that no numerical errors are introduced in the conversion process.

Once all the simulations functions invoked during each time step were migrated and verified, the host code was removed to ensure that our simulation runs purely on the GPU and no host code with similar function could be called accidentally.

III. REDUCING DATA DEPENDENCIES

The GPU programming paradigm requires us to minimise (and ideally remove) dependencies between workitems so that separate instances of our kernels can run independently without the need for costly synchronisation.

In its original form, the propagate function pushes density values outwardly towards neighbouring cells and stores the intermediate result in a shared temporary buffer. Our version of the function will instead read values inwards from neighbouring cells so that only the thread iterating over the current cell will ever write to the memory location belonging to that cell (Bella, et al. 2002).

IV. MAXIMISING THE NUMBER OF POTENTIALLY PARALLELISABLE REGIONS

The reference C implementation of the Boltzmann simulation contains four functions and an average velocity calculation all invoked in the same order for each time step in the simulation. I.e.:

Acceleration → Propagation → Rebound → Collision → Average velocity calculation

A direct conversion of these C functions into OpenCL kernels translates into five different kernels each invoked 10000 times resulting in the submission of 50000 OpenCL commands to the OpenCL command queue.

Each kernel launch has a cost associated with it, so to speed up our program we need to reduce the number of kernels required by the simulation to just one, which would in turn reduce the number of OpenCL commands to just 10000, e.g. one command per iteration. Further study reveals that this amalgamation of kernels into one can be achieved if we first reorder the execution of the kernels as follows:

Propagation → Rebound → Collision → Average velocity calculation → Acceleration

Please note that this approach requires us to apply a one-off acceleration step at the beginning of the simulation either when the grid is being initialized on the host or by running a one-off acceleration kernel before the main compound kernel is launched. In addition to this, the main kernel must omit the very last acceleration step at the end of the simulation. We have reduced the cost of the comparison at each time step to determine whether the executing kernel belongs to the last iteration by counting down to zero, instead of increasing the iteration counter and comparing at that point.

When we combine this new kernel ordering approach with the data dependency reduction from the previous section we find that the kernel can now be easily combined into just one. Furthermore, the merged kernel distributes the workload more evenly amongst threads because the conditional statement within the inner loop now performs some computation when obstacles are both present and absent. Furthermore, to limit divergence in conditional statements, we always perform both calculations irrespective of whether blocks are present or absent. Divergence is therefore limited to the statement that selects the appropriate result depending on whether the cell was blocked or otherwise free from obstruction.

V. Memory access

A. *Coalescence*

One of the biggest speeds boosts to our program has been achieved by ensuring coalesced access to global memory (David and Hwu 2010). In the reference implementation of the Boltzmann simulation, all the speeds belonging to one cell are kept in a single structure, with the enclosing lattice containing $m \times n$ of these structures; an arrangement commonly known as an array of structures (AoS) (Wittmann, et al. 2011). This arrangement is ideal for the collision calculations but suboptimal for our inverted propagation step because workitems do not access contiguous memory addresses simultaneously. In other words, each workitem is accessing a memory address 9 elements apart from the previous workitem.

To improve access to global memory, we have logically rearranged our data into a structure of arrays (SoA) where all the speed vectors pointing in the same direction are kept in a single contiguous array. In our case, this creates nine arrays, one per directional speed, of $m \times n$ elements. For simplicity, we have kept the memory allocation functions largely unchanged and used C pre-processor macros to dereference memory using a SoA memory access pattern.

B. *Locality*

A second important factor affecting the performance of our program is the number of global memory accesses. Where possible, our aim should be to increase reuse of locally available data to limit the number of global memory accesses. We have achieved this by further integrating the combined kernel so that once data is loaded into local and private memory from global memory, and it is then reused by each of the logical Boltzmann simulation steps requiring access to that data without incurring the penalty of a global memory access instruction. In other words, speeds are copied into private memory and subsequently propagated, collided and reduced without further global memory accesses until they need to be saved.

VI. Reduction

The process of calculating the average velocity per time step is somewhat different on a GPU than a CPU because the innate parallelism of the GPU does not allows us to produce a single value as a result of the reduction operation. Instead we must have each workgroup on the GPU reduce a part of the larger grid and have the CPU recombine the per-workgroup results into a single figure (David and Hwu 2010).

Reduction on the GPU is performed by way of iterative binary addition. The process of reduction can be thought of as a level by level aggregation on an inverted binary tree, starting at the leaf nodes and working, level by level, all the way up to the root. To begin with the inner nodes in the tree are empty and only the leaf nodes, which correspond to elements in the grid, are populated. As we iterate upwards from the leaf nodes, the inner nodes are assigned the sum of their children. The process is then repeated until the root of the tree is reached resulting in a single value. To make the reduction process work correctly and efficiently, the workgroup size must be a power of two and the results of every iteration arranged in such a way so as to avoid bank conflicts.

To speed up overall program execution, we do not send the reduced average velocity back to the host at every time step but rather compute and store all velocities on the GPU until all 10000 iterations are complete. Only after the 10000 iteration has completed do we send all the average velocities back to the host.

The reduction process happens in two stages. The first stage takes places right after the calculation of the new collision values and results in one aggregate value per workgroup. The second stage takes place at the very end of the simulation and is responsible for combining the different workgroup values belonging to the same iteration into a single scalar value.

Note that we have unwound the last loop using a few lines of code from the NVidia SDK reduction example to avoid unnecessary synchronisation within the same warp (32 threads). Because this optimisation applies specifically to NVidia architectures and it is not a portable technique, our code includes both the rolled and unrolled loops to allow the program to run on CPUs and GPUs from other vendors.

VII. Strategies for improving parallel performance

A. *Precalculating invariants*

A number of operations in the original reference implementation perform repeated invariant calculations for each time step in the simulation. Taking advantage of the fact that the kernel is compiled on request from the host code, these invariant calculations can be performed once and then passed to the OpenCL runtime as compile time parameters for the kernel as C pre-processor macros.

B. *Increasing occupancy*

As discussed early, launching a kernel incurs a cost. One way of reducing this cost is to reduce the number of kernels. A second approach once the first one has been exhausted is to have each instance of the kernel perform more work for each invocation. That is to say, each thread can perform calculations on more than one cell each time it is invoked.

In our implementation, each kernel instance will perform calculations on nine cells per invocation using a stride access pattern. E.g. in a workgroup consisting of 512 workitems, each workitem will first access a contagious memory address, perform the required computations and then move on to the next cell, found 512 positions apart, until all nine cells in the stride have been processed.

In practice, this involves ensuring that every single compute unit in the device is simultaneously occupied and that there are at least four workitems per work thread so that latency in global memory accesses and branching can be hidden away.

For our 200 x 300 cell grid size we have chosen to the following parameters:

TABLE I. OPENCL ND RANGE PARAMETERS

Parameter	Value	Description
Global	7168	Total number of workitems per iteration.
Local	512	Number of workitems in the workgroup. A power of two to simplify arithmetic and reduction.
Workload per workitem	9	Number of distinct cells processed by each workitem. Calculated so that "Local x Workload x Workgroups" or "Global x Workload" is as close as possible to the total number of cells in the grid and divides into the number of compute units.
Number of workgroups	14	Number of workgroups given by Global/Local. Ideally a multiple of the number of compute units on the OpenCL device to ensure optimal occupancy.

C. *Reducing branching*

Idle time between time steps is minimised by distributing the workload evenly across workitems and avoiding divergence. In the case of the rebound and collision steps this involves performing the calculations irrespective of the branch and selecting the appropriate result at the very end of the calculation.

VIII. OUTLINE OF THE PARALLEL ALGORITHM

A. *Host code*

The host code in our program is largely made up of boilerplate code to set up the environment, allocate and initialise memory and to obtain handles to an OpenCL device and associated command queue.

Once the environment has been successfully configured, our host program performs the following operations:

- Transfers the memory buffers to the device.
- Submits a one-off acceleration kernel command to the OpenCL command queue.
- Submits 10000 commands to execute the combined (propagate/collide/reduce) kernel making sure that the buffer parameters are swapped between odd and even time steps by calling *clSetKernelArg(...)*.
- Submits a one-off final reduction kernel command to the OpenCL command queue.
- Waits for the simulation to complete and writes the results to file.

B. *Device code*

Out program uses three kernels in total. The main kernel, which combines the propagate, collide and reduce functions, being the more complex of the three. Out of the remaining two kernels, one is run once at the beginning of the simulation to accelerate the particles and the other once at the very end of the simulation to complete the average velocity reduction job partially performed by the main kernel.

For each iteration of the simulation, our main kernel performs the following actions:

- Initialised reduction variable to 0.
- Then for each stride (or number of cells to be processed by the workitem):
    - Pull in speeds belonging to neighbouring cells from the source buffer and store the values in a private array A of speeds. This effectively amounts to performing the propagation step.
    - Take the private copy of the propagated speeds and perform a rebound operation storing the result in another private array B of speeds.
    - Perform collision calculation on private array of speeds A.
    - Store one of private arrays A or B in the output buffer in global memory depending on whether there an obstacle is absent or present respectively from that cell in the grid.
    - Add the average speed from private array A or B, depending on the absence or present of an obstacle, to the reduction variable.
    - Accelerate the cell in the output buffer in global memory if it is in the first column.
- Synchronise workgroup and perform further average velocity reduction to obtain per-workgroup partial averaged velocity.

IX. ISSUES NOT CONSIDERED IN THIS DOCUMENT

A. *Vectorization*

Since the CUDA architecture is natively scalar (NVidia Corporation 2011) and vectorization does not bring any performance benefit on this architecture, we have chosen not to use vector data types and vector instructions in our code.

B. *Numeric constant conversion*

Although the NVidia best practices guide suggest that kernel performance can be improved by explicitly adding the "f" suffix to numeric constants as an indication to the compiler to avoid unnecessary type promotion in single precision expression, our experiments have shown no actual difference between those programs where single precision constants had the constant appended and those that did not. As a result we have omitted the suffix to ensure the program runs correctly and accurately when compiled with double precision support.

X. Results

Our program has been written to allow use of either single or double precision floating point arithmetic by simply recompiling the code. We have produced results in double precision floating point to verify that the output of our program matches that of the reference implementation and subsequently switched to single precision floating point in order to reduce program execution time.

Using single precision floating point arithmetic **our final program achieved 10000 iterations in 0.57 seconds** but accuracy was reduced to three decimal places. Using double precision floating point arithmetic, to trade speed for precision, the Reynolds number was identical to that generated by the reference serial implementation but the execution time increased to 1.14 seconds.

Please note that the average velocities were not exact, but near identical, differing only in the last digit. This is not unexpected since instances of a program compiled with different compilers do usually yield marginally different results. Indeed, this is also the case for instances of a program compiled with the different compiler using different flags.

TABLE II. BOLTZMANN SIMULATION RUNTIME.

Floating point precision	Wall time in seconds	GPU average velocity	Reynolds number
Single	**0.57**	3.569530323148E-02	2.641452789307E+01
Double	1.14	3.569878903889E-02	2.641710388878E+01

TABLE III. EXECUTION TIME AND SPEEDUP RESULTS VS OPTIMISED SERIAL CODE

Program type	Wall time in seconds	Speedup
Optimised Serial code	86.26	1.00
Parallel OpenMP (4 threads)	34.07	2.53
Parallel MPI (16 cores)	4.27	20.21
OpenCL double precision (1 GPU)	1.14	75.66
OpenCL single precision (1 GPU)	**0.57**	151.33

References

[1] B. Chapman, G. Jost and R. van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, Massachusetts: The MIT Press, 2007.

[2] G. Bella, S. Filippone, N. Rossi and S. Ubertini, “Using openMP on a hydrodynamic lattice-Boltzmann code,” in *Proceedings of the Fourth European Workshop on OpenMP*, Roma, 2002.

[3] K. David and W.-m. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington: Morgan Kaufmann Publishers, 2010, p. 256.

[4] NVidia Corporation, Guide, OpenCL Best Practices, 2.2 ed., NVidia Corporation, 2011, p. 54.

[5] M. Wittmann, T. Zeiser, G. Hager and G. Wellein, “Comparison of different Propagation Steps for the Lattice Boltzmann Method,” *submitted to Computers & Mathematics with Applications,* p. 18, 2011.

[6] Khronos OpenCL Working Group, The OpenCL Specification, 44 ed., A. Munshi, Ed., Khronos OpenCL Working Group, 2011.

[7] M. Scarpino, OpenCL in Action, How to accelerate graphics and computation, Manning Publications Co., 2011, p. 456.

[8] A. Munshi, B. Gaster, T. G. Mattson, J. Fung and D. Ginsburg, OpenCL Programming Guide, Addison Wesley, 2011, p. 648.