Distributed Memory Lattice Boltzmann

Alexander Carolan - AC17588
Computer Science
University Of Bristol

I. Introduction

This report describes the process of further optimising a lattice Boltzmann fluid simulator, operating in 2 dimensions with 9 speeds. The report will first focus on the conversion to a distributed memory homogeneous architecture, using 8 CPUs, and then the conversion to a distributed memory heterogeneous architecture, using 1 CPU and 1 GPU. Throughout the report each optimisation is detailed in the form of run times, averaged across 10 runs on three different grid sizes. The largest grid of 1024×1024 is more representative of simulations typically performed in the high performance setting and thus will be the focus of my discussion.

II. Base Optimisation

As the basis for both the homogeneous and heterogeneous versions of the lattice Boltzmann fluid simulator, I will be using the previously optimised version discussed in the first report. However both the homogeneous and heterogeneous versions will lack its parallelisation, by virtue of employing their own, and the heterogeneous version will lack its vectorisation, by virtue of employing its own. As shown in Table I though I was able to make one further improvement to the base version, resulting in a speedup of 1.09X for both the serial and vector varieties on the largest grid.

TABLE I
Base optimisation comparisons

Optimisation	**Grid Size**					
	128 × 128		256 × 256		1024 × 1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Serial original	11.46	1.00	94.12	1.00	406.37	1.00
Serial improved	9.72	1.18	83.86	1.12	374.23	1.09
Vector original	7.54	1.00	68.34	1.00	327.26	1.00
Vector improved	5.95	1.27	60.63	1.13	301.17	1.09

[1] Speedup values were calculated against original speeds.

This improvement consisted of replacing the use of the modulo operator `%` with the ternary operator `?` in the calculation of neighbouring cells. The modulo operator is akin to the division operator, a relatively expensive instruction to execute, whereas the ternary operator is akin to a conditional branch, a relatively inexpensive instruction to execute. As this calculation is performed in every iteration of the main kernel's fused loop, it resulted in a significant speedup despite being such a minor change.

III. Homogeneous Optimisation

Distributed memory homogeneous programs utilize multiple processors of similar kind, each with their own distinct memory. As such when one processor requires data located in another processor's memory, communication between the two is necessary. This communication has an inherent cost associated with it and therefore should be minimized as much as possible. Despite this the ability to parallelise code across not just one node but an entire cluster, allows for significant speedups. As shown in Table II utilising 112 cores from 8 CPUs in 4 nodes resulted in an enormous speedup of 54.17X.

TABLE II
Homogeneous optimisation comparisons

Optimisation	**Grid Size**					
	128 × 128		256 × 256		1024 × 1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Parallelisation	1.47	4.05	9.96	6.09	7.56	39.84
Even distribution	1.25	4.76	4.92	12.32	5.56	54.17

[1] Speedup values were calculated against vector improved speeds.
[2] 112 cores were used for all experiments.

The first thing to consider in this conversion process is the method by which to decompose the grid, so that it can be distributed amongst the different processes. As the grid in this case is represented in a row major fashion, it's only natural to decompose it down into rows. This not only reduces the amount of communication required when compared to a tile decomposition, as communication is only required between processes above and below but not left and right in the grid. It also results in better data access patterns when compared to a column decomposition, as each row is already a contiguous unit of memory.

Once this was settled, I began the conversion by creating global data structures for the master process to collate on, and local data structures for every process to compute on. Including an additional two rows in the local grids known as halo regions, to store portions of the grid required from neighbouring processes, which must be communicated with each time step. As each process can have a variable number of rows, it became simpler to store the local total velocity rather than local average velocity.

Initially the local data structures were given an uneven distribution of rows, with any remaining rows assigned to the last process. They were also all initialised on the master process and then communicated to every other process using the

`MPI_Ssend` and `MPI_Recv` functions. With synchronous communication being used to ensure that the communication had completed, and the buffers were safe to reuse before computation began.

Subsequently the indexing used in the loops of both the acceleration kernel and the main kernel needed modifying, so that they iterate over the local grids and take into account the halo regions. The main kernel proved itself to be trivial however the acceleration kernel proved itself to be more nuanced. Whilst only one process can contain the accelerated row in its local grid, it is possible for two other processes to contain it in their halo regions. If this is the case, then these processes will require synchronisation before the main kernel can begin. Instead of using communication between the processes which would come at a significant cost, I used a conditional statement to enable these processes to accelerate the row themselves and so stay synchronised.

From here the halo exchange was implemented at the end of each time step using the `MPI_Sendrecv` function. In order to avoid deadlock each process first sends to the below process and receives from the above process and then vice versa. Now the local data structures were being computed successfully, they could be collated into the global data structures. For the local grids this was done using the `MPI_Ssend` and `MPI_Recv` functions. Whereas for the local total velocities this was done using the `MPI_Reduce` function, creating a global total velocity whose values were then divided by the total number of cells, to yield a global average velocity.

Finally I performed a couple of optimisations, the first of which was to distribute the data initialisation amongst the processes. This obviously had no effect on the compute time, but did result in a significant speedup of 27.36X in initialisation time for the largest grid. The second of which was to give the local data structures an even distribution of rows, by assigning each subsequent process an additional row of the remaining rows. This resulted in a significant speedup, especially for the middle grid with it resulting in a speedup of 12.32X, with it previously running slower than the largest grid, owing to the fact that it had the largest amount of remaining rows. Thus this optimisation acts to make the run times more consistent as the number of cores is varied.

IV. Homogeneous Analysis

This analysis begins with a scaling graph comparing the speedup of the lattice Boltzmann program on each of the three grid sizes, starting at one core and ending at 112 cores. This was achieved using the `MPI_Comm_split` function to create two communicators, one for the active processes and one for the inactive processes. Furthermore the slurm configuration flag `distribution` was used to ensure each successive process was assigned to the closest socket available. As you can see in Figure 1 initially the middle grid sees the largest speedup, then the largest grid and lastly the smallest grid. This trend continues up until the 28th core, the total number of cores on a single node in Blue Crystal phase 4.

After which both the middle grid and the smallest grid experience a sharp decrease, resulting from the network interconnect bottleneck in conjunction with the diminishing returns available, as the ratio of halo cells to actual cells increases. This results in the smallest grid flattening out a roughly 6X speedup, and the middle grid plateauing at a roughly 20X speedup. In contrast the largest grid continues to speedup and even experiences a massive increase after the 36th core, likely a result of the grid now fitting inside the cache. Not suffering from a high ratio of halo cells to actual cells, it climbs to a roughly 80X speedup. Overall I would characterise the scaling characteristics of this program as that of sub-liner plateau.

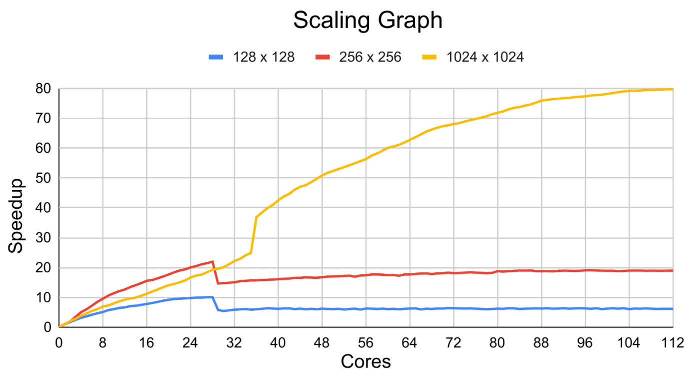

Fig. 1. Homogeneous Lattice Boltzmann scaling graph.

Now I turn to a roofline model I generated myself of the homogeneous lattice Boltzmann code. This was done by manually analysing the code of the main kernel, to produce a FLOPs performed per iteration of 145 and a Bytes loaded or stored per iteration of 73. From here I calculated the operational intensity to be at a value of 2.04 FLOP/byte and the performance to be at a vale of 547 GFLOP/s. As you can see from Figure 2, this places the lattice Boltzmann program safely in the memory bandwidth bound region, being significantly far from the DRAM ridge point. However it also places it above the theoretical maximum DRAM bandwidth, entering it into the L3 cache territory.

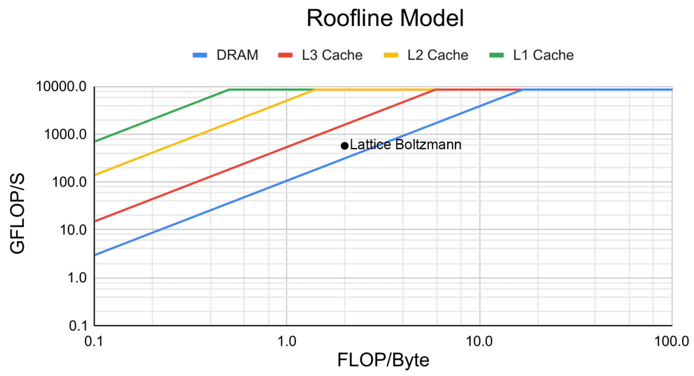

Fig. 2. Homogeneous Lattice Boltzmann roofline model.

V. Heterogeneous Optimisation

Distributed memory heterogeneous programs utilize multiple processors of different kinds, each with their own distinct advantages and disadvantages. Whilst latency optimized CPUs perform better on dynamic serial tasks, throughput optimized GPUs perform better on static parallel tasks. Whenever a task is transferred from one processor to the other, communication between the two is required. Once again this communication has an inherent cost associated with it and should therefore be minimized. However the ability to utilize these architectural differences, allows for significant speedups. As shown in Table III utilising a GPU resulted in a significant speedup of 17.10X.

TABLE III
Heterogeneous optimisation comparisons

Optimisation	**Grid Size**					
	128 × 128		256 × 256		1024 × 1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Parallelisation	11.92	0.82	67.42	1.24	180.34	2.08
Pointer swap	1.78	5.46	8.59	9.76	21.88	17.10

[1] Speedup values were calculated against serial improved speeds.

The first step in the conversion process was to construct the necessary environment for executing a heterogeneous program, in which the CPU acts as the host and the GPU acts as the device. This included selecting the GPU device, creating a context within which it can execute, creating a queue from which it can execute, creating buffers for global data structures and setting any kernel arguments. Once this was completed I moved the kernels to a separate file, which is then compiled just in time so that it remains platform independent. As each iteration of a kernel's loop is treated as a separate work item, they could be removed and replaced with the `get_global_id` function which returns a unique work item identifier.

Modifications to the kernels were necessary in order to make them compatible with the heterogeneous environment. This included converting the grids structure of arrays data layout, to an equivalent single contiguous array data layout, as structures containing pointers are not allowed. It was important to maintain this data layout in order to fully leverage the architecture of the GPU. The GPU consisting of multiple compute units, themselves consisting of multiple processing elements, which can take advantage of a technique known as memory coalescing, similar in nature to vectorisation. Memory coalescing is the combination of multiple memory accesses into a single memory access, but is dependent on all the necessary data being contained in a single contiguous aligned unit of memory.

Alongside this the average velocity with each time step could no longer be computed in the main kernel, as each work item only has access to the velocity of an individual cell. Furthermore function returns are not allowed due to the host and device having distinct memories, so in order to solve this a new buffer was created for an iteration velocity array, in which each work item stores its individual velocity. This is then read back to the host with each time step, which then computes the average as before.

Finally I performed one optimisation, which was to move the pointer swap from occurring on the host to occurring on the device. This resulted in a dramatic speedup, especially for the smallest grid with it resulting in a speedup of 5.46X, with it previously running slower than the serial version. As a throughput optimized architecture the ratio of communication to computation, is a pivotal factor in the performance of any program executed on it. This only gets greater the smaller the grid gets, as there are less work items compared to processing elements and so the more frequently it has to communicate with the host.

VI. Heterogeneous Analysis

This analysis unlike the previous one does not begin with a scaling graph, owing to the fact that the number of cores is fixed on a GPU. I did however produce another roofline model, this time of the heterogeneous lattice Boltzmann code. This was done in the exact same manner as earlier, and once again produced an operational intensity of 2.04 FLOP/byte, but instead produced a performance of 139 GFLOP/s. As you can see from Figure 3, similar to before this places the lattice Boltzmann program safely in the memory bandwidth bound region, being significantly far from the DRAM ridge point. Unlike before however this places it just below the theoretical maximum DRAM bandwidth.

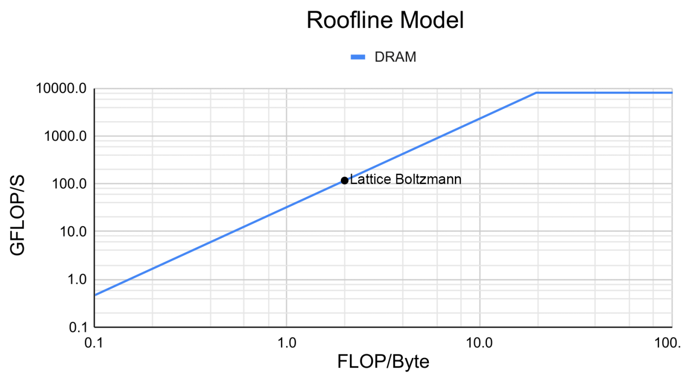

Fig. 3. Heterogeneous Lattice Boltzmann roofline model.

VII. Conclusion

In conclusion I consider this optimisation of a base lattice Boltzmann code a great success, with a total speed up from completely unoptimised to completely optimised of 58.86X and 18.57X for the homogeneous and heterogeneous code respectively, on the largest grid. There is always room for improvement though and if I were to continue this, I would take inspiration from Obrecht *et el.* [1] who detail a multi-GPU implementation.

References

[1] https://hal.archives-ouvertes.fr/hal-00731106/document
[2] https://uob-hpc.github.io/BabelStream/