Shared Memory Lattice Boltzmann

Alexander Carolan - AC17588
Computer Science
University Of Bristol

I. Introduction

This report describes the process of optimising an initial lattice Boltzmann fluid simulator, operating in 2 dimensions with 9 speeds. 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. Compiler Optimisation

In optimising compilation you face two choices, a choice of compiler and a choice of flags to use in conjunction with the compiler. Table I compares GCC version 9.1.0 with Intel version 18.0.3 using a variety of flags. Flags O0 through Ofast increasingly apply general optimisation techniques, whereas the mtune flag applies specific optimisation techniques for a given processor architecture. Interestingly the GCC and Intel compilers differ in the aggressiveness at which they apply optimisations, Intel produces a large speedup from O1 to O2 of 1.38X whereas GCC produces a large speedup from O3 to Ofast of 1.08X.

TABLE I
Compiler optimisation comparisons

Compiler	**Flags**	**Grid Size**					
		128×128		256×256		1024×1024	
		Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
GCC	O0	152.78	1.00	1282.37	1.00	5025.95	1.00
GCC	O1	39.52	3.87	319.94	4.01	1323.07	3.80
GCC	O2	39.43	3.87	317.72	4.04	1307.87	3.84
GCC	O3	37.99	4.02	306.55	4.18	1287.50	3.90
GCC	Ofast	33.24	4.60	268.46	4.78	1189.11	4.23
GCC	mtune	33.24	4.60	268.44	4.78	1189.00	4.23
Intel	O0	151.30	1.01	1213.09	1.06	4948.69	1.02
Intel	O1	44.74	3.41	360.57	3.57	1479.97	3.40
Intel	O2	30.98	4.93	249.48	5.14	1073.34	4.68
Intel	O3	30.31	5.04	244.62	5.24	1054.72	4.77
Intel	Ofast	30.29	5.04	242.86	5.28	1047.67	4.80
Intel	mtune	30.28	5.05	242.86	5.28	1047.58	4.80

[1] Speedup values were calculated against slowest time for each grid.
[2] The mtune flag was used in conjunction with Ofast.

Table I is by no means exhaustive but it does cover the major optimisation flags supported by both compilers and clearly demonstrates an edge across all three grid sizes for the Intel compiler. The most optimised Intel compilation producing a speedup of 1.13X relative to the most optimised GCC compilation on the largest grid. For this reason and also for the option to use Intel's aligned memory allocation, I chose to use the Intel compiler throughout the rest of the optimisation process.

III. Serial Optimisation

As lattice Boltzmann methods are primarily memory bandwidth bound rather than compute bound, the focus of my serial optimisations was on reducing the demand on memory. This involved not only reducing the total amount of transfers to and from memory but also the size of the data being transferred. As shown in Table II the greater persistence of data in the caches in conjunction with their higher bandwidths, resulted in an overall speedup of 1.16X for the largest grid.

TABLE II
Serial optimisation comparisons

Optimisation	**Grid Size**					
	128×128		256×256		1024×1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Loop fusion	29.39	1.03	237.05	1.02	1046.62	1.00
Pointer swap	27.24	1.11	219.97	1.10	906.86	1.16
Obstacles	26.87	1.13	218.86	1.11	905.43	1.16
Arithmetic	26.89	1.13	218.78	1.11	905.24	1.16

[1] Speedup values were calculated against previous section's best.

The first step in this process was to perform loop fusion on the four kernels: propagate, rebound, collide and average velocity, that shared a common nested for loop that iterated over the entire grid. The first thing to note is that all the essential computations of the average velocity kernel were already being computed by the collide kernel and so it can be removed entirely. Fusing the rebound and collide kernels was a simple process, as they are conditionally independent on whether or not the current cell contains an obstacle and can simply be combined into an if else statement.

Fusing the propagate kernel was more complex due to a data dependency between it and the collide/rebound kernel. The original kernels utilised a *source* $\rightarrow$ *temporary* $\rightarrow$ *source* read and write design for modifying the cells in the grid. However in each iteration of the loop the propagate kernel requires the values of neighbouring cells at the beginning of the time step but the collide/rebound kernel modifies those with values at the end of the time step. To overcome this a *source* $\rightarrow$ *temporary* $\rightarrow$ *destination* read and write design was adopted, noting however that *temporary* need only be a single cell not an entire grid, greatly reducing the memory

requirements. Only at the end of the time step are the values of *source* assigned to those of *destination*.

The next logical step was to replace the additional nested for loop assigning the values of *source* to those of *destination*, with a much simpler and significantly more memory efficient pointer swap. As can be seen in Table II this enabled the loop fused kernels to provide a major speedup across all three grid sizes. From here I focused on reducing the size of the data being transferred to and from memory, modifying the obstacles array data type which only needs to represent Boolean values, from being `int` (4 bytes) to `char` (1 byte), resulting in a size reduction of 4X.

Finally I turned my attention simplifying the computation by removing unnecessary intermediary constants and converting divisions to multiplications, a less expensive instruction for the processor to execute. From the results in Table II it is clear that the compiler was already performing these optimisations or that being a memory bandwidth bound program, they had little to no significant effect on run time.

IV. Vectorisation

Vectorisation is the process of converting a scalar implementation in which a single instruction operates on a single piece of data at a time, into a vector implementation in which a single instruction operates on multiple pieces of data (vector) at a time. As shown in Table III the reduction of singular instruction executions and singular data transfers resulted in an impressive speedup of 2.83X. This closely matches the reported speedup of 2.69X produced using Intel's profiler with vectorisation reports enabled.

TABLE III
VECTORISATION OPTIMISATION COMPARISONS

Optimisation	**Grid Size**					
	128 × 128		256 × 256		1024 × 1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Vectorisation	9.31	2.89	66.05	3.31	320.24	2.83
Pointer restriction	9.21	2.91	65.78	3.33	319.69	2.83
Memory alignment	7.52	3.57	65.12	3.36	320.23	2.83
Modulo assumption	7.40	3.61	68.79	3.18	332.86	2.72

[1] Speedup values were calculated against the previous section's best.
[2] The march flag was used in conjunction with previous flags.

The benefit of having already performed loop fusion is that it greatly simplifies the vectorisation process, with there being only one loop to vectorise. The next step in the process was to convert the program from representing the grid with an array of structures data layout to a structure of arrays data layout, which results in more memory efficient data access patterns for vectorised code. After this was completed it was simply a matter of wrapping the inner loop of the time step kernel with the following openmp directive `#pragma omp simd reduction(+: tot_cells, tot_velocity)`. This instructs the compiler to vectorise the loop, whilst the `reduction` clause also specifies that the `tot_cells` and `tot_velocity` variables must be summed across all iterations of the loop.

From here I added the type qualifiers `restrict` and `const` to any variables meeting those requirements. This informs the compiler that pointers won't alias and that variables won't be modified, which can aid in vectorisation process. As seen in Table III it had a small but noticeable effect on the speedup, especially at the lower grid sizes. Another alteration that can aid in the vectorisation process is the allocation of variables on the heap in alignment with the 64 byte cache lines of the processor. I achieved this using Intel's `_mm_malloc` function to allocate them in conjunction with their `_mm_free` function to free them.

Interestingly this caused a large speedup for the smallest grid, a moderate speedup for the middle grid but effectively no speed up for the largest grid. The last optimisation I attempted but ultimately abandoned, was to hint to the compiler that the maximum iterations of both loops was cleanly divisible by 16, using the `_assume` function. This did result in a speedup for the smallest grid, however it caused a reduction in speedup for both the middle grid and the largest grid, so in accordance with my statement in the introduction I decided to not implement it.

V. Parallelisation

Parallelisation is the process of converting a serial implementation in which a single processor executes a single program, into a parallel implementation in which multiple processors execute a single program or a portion of that program. As shown in Table IV the conversion from serial computation to parallel computation utilising 28 cores resulted in an enormous speedup of 13.57X.

TABLE IV
PARALLELISATION OPTIMISATION COMPARISONS

Optimisation	**Grid Size**					
	128 × 128		256 × 256		1024 × 1024	
	Time/s	*Speedup*	*Time/s*	*Speedup*	*Time/s*	*Speedup*
Parallelisation	1.71	4.32	6.98	9.32	25.56	12.51
Data initialisation	1.63	4.54	6.34	10.27	25.36	12.60
Thread variables	1.10	6.73	5.38	12.10	23.55	13.57

[1] Speedup values were calculated against the previous section's best.
[2] The qopenmp flag was used in conjunction with previous flags.
[3] 28 cores were used for all experiments.

Similarly to vectorisation this was simply a matter of wrapping the outer loop of the time step kernel with the following opnemp directive `#pragma omp parallel for reduction(+:tot_cells, tot_velocity) num_threads(THREADS)`. Where the `reduction` clause serves the same purpose as in vectorisation and the `num_threads` clause specifies the number of parallel threads to execute the loop with. As a result of the scope in which the variables of the time step kernel were declared no `private` and `shared` clauses were necessary. This optimisation alone gave the vast majority of the speedup

achieved at 12.51X. Interestingly appears to scale with the size of the grid, as a result of larger grids being more and more memory bandwidth bound.

Although the program had become significantly faster at this point it had also become significantly more varied in its run time. So in order to correct for that I made the program NUMA (Non-uniform memory access) aware, by not only computing on the data in parallel but initialising the data in parallel, so that each thread computes on the same data it initialised. The Blue Crystal phase 4 supercomputer on which this program was run, has two sockets per node each with their own DRAM and cache hierarchies. If a thread on one socket requires data from the other sockets DRAM it must go through the socket interconnect, which is significantly slower than accessing its own DRAM and thus presents a large bottleneck.

This alone wasn't enough to significantly increase speedup however as can be seen in Table IV. The reason being is that the threads themselves weren't pinned to specific processor cores and were free to move around during execution. To fix this I set the `OMP_PROC_BIND` environment variable to true at the time of execution, furthermore I added the `proc_bind` clause to the parallel directives mentioned earlier and set it to close. This causes threads to be assigned to cores as close to the master thread as possible, the reason I chose this option rather than spread is so that threads working on close areas of the grid are assigned to close cores and thus hopefully results in better data locality. This does have the downside though of not evenly distributing threads across sockets and maximizing all available bandwidth.

VI. 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 28 cores. As you can see in Figure 1 the program initially scales very similarly across all three grids, however after the 3rd core they begin to diverge. This process continues all the way up to the 14th core, the total number of cores on a single socket in Blue Crystal phase 4. Interestingly after the 11th core the largest grid actually experiences less of a speed up than the smallest grid, likely a result of it experiencing a greater bottleneck from the limited memory bandwidth. At 14 cores it is the middle grid that experiences the greatest speed up, after that the smallest grid and lastly the largest grid.

After the 14th core grids of all sizes experience a dip in speed up, resulting from the socket interconnect bottleneck mentioned earlier. However it is only the smallest grid that doesn't rebound and benefit from the additional cores and extra memory bandwidth, instead plateauing at a roughly 7X speedup. Both the middle grid and largest grid continue to experience speed ups all the way up to 28 cores, with the largest grid superseding the middle grid after the 19th core. The middle grid finishing at a roughly 12X speed up and the largest grid finishing at a roughly 14X speedup. Overall I would characterise the scaling characteristics of this program as that of sub-liner plateau.

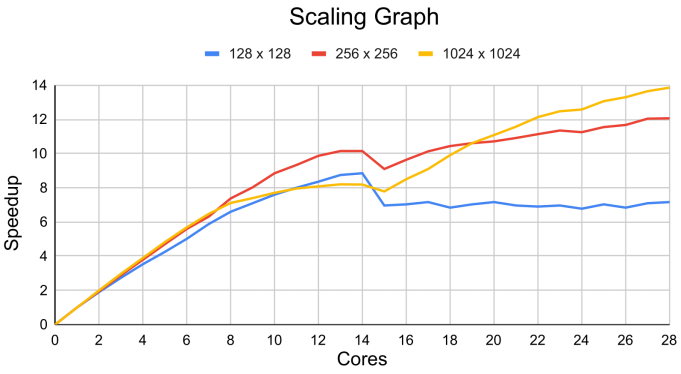

Fig. 1. Lattice Boltzmann scaling graph.

Now I turn to a roofline model I generated myself of the parallel vectorized 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 value of 129 GFLOP/s. As you can see from Figure 2, this safely places the lattice Boltzmann program 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.

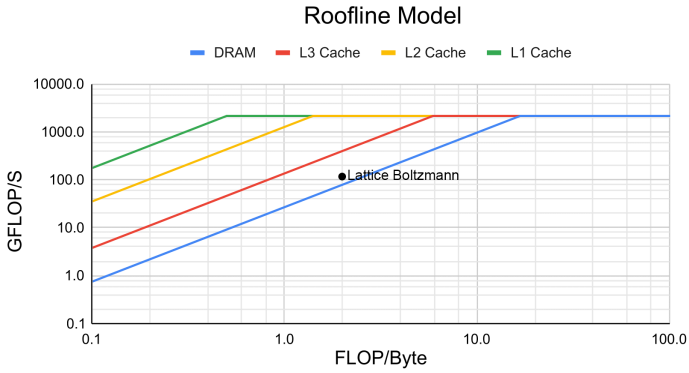

Fig. 2. Lattice Boltzmann roofline model.

VII. Conclusion

In conclusion I consider this optimisation of an initial lattice Boltzmann code a great success, with a total speed up from completely unoptimised to completely optimised of 213.42X for the largest grid. There is always room for improvement though and if I were to continue this, I would take inspiration from Pananilath *et el.* [1] who discuss its similarity to other stencil programs and utilise time-tiling to further optimise.

References

[1] https://dl.acm.org/doi/pdf/10.1145/2739047
[2] https://www.cs.virginia.edu/stream/ref.html