Optimisations of the Lattice Boltzmann Simulation on a Single HPC Node

Paolo Mura
rk19763

I. Introduction

This report describes several layers of optimisations applied to the Lattice Boltzmann simulation, which was originally written in serial C code. I first improved the efficiency of the serial code by simplifying redundant operations and complex arithmetic. I then converted the cells data structure from an Array of Structures to a Structure of Arrays, allowing for use of vectorisation. Ultimately, I used OpenMP to parallelise the solution across all 28 cores of a node on Blue Crystal phase 4. The final parallel version achieved up to 94X speedup when compared with the original code and 57X increase in GFLOPS compared with the optimised serial version.

Throughout the report, I shall use the following abbreviations: LBM (Lattice Boltzmann), BC4 (Blue Crystal phase 4).

II. Serial Optimisations

A. Compiler Optimisations

The default compiler on BC4 is gcc v4.8.5. I loaded the latest version of the gcc compiler available on BC4 (v10.4.0) as well as the latest Intel compiler (icc v19.1.3.304). From here, I built the original unmodified LBM code using each compiler on several grid sizes to compare performance.

I also chose to test a variety of compiler flags with each one. These include -O3, -Ofast and -mtune=native (gcc) or -fast (Intel). Since flags like -Ofast introduce aggressive and potentially unsafe math optimisations, I ran the make check command to verify that it didn't significantly alter calculation results. The Intel compiler does not have a -mtune=native flag, so I chose to use the -fast flag instead, which turns on the equivalent -xHost option as well as some others. After running the simulations, I plotted bar charts to visualise the comparison of results.

It is evident that the Intel compiler resulted in the best optimisations for this problem. It varied depending on grid size as to whether the -O3 or -Ofast flag gave best optimisations. Going forward, I stuck with the -O3 flag and Intel compiler.

B. Loop Fusion

Use of the gcc profiler revealed that 71.24% of the runtime was being spent in the collision function and when combined with av_velocity and propagate, contributed approximately 100% of the runtime.

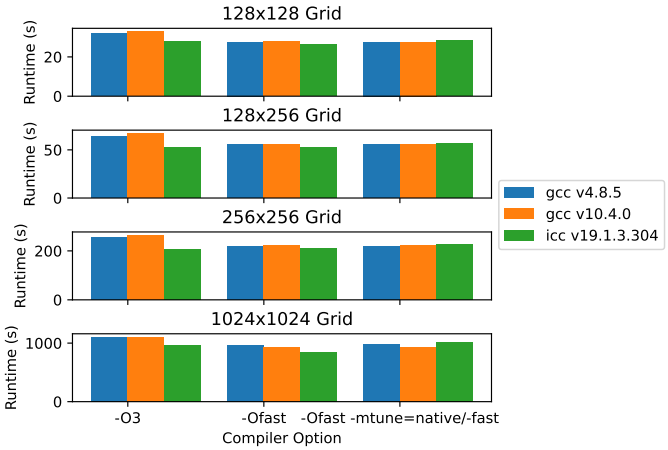

Fig. 1. Bar chart showing the run times of the unoptimised code when built using different compilers and flags.

The first manual optimisation I made was fusing the loops contained within each of the main functions (accelerate_flow, propagate, rebound, collision and av_velocity) into the timestep function.

This meant that the code only iterated over the grid once per time step, improving efficiency by reducing the number of data transfer operations. In other words, data is loaded once, operated on, then written once per time step instead of loaded and written multiple times between each set of operations.

To avoid unnecessary copying of data between the cells and tmp_cells grids, I refactored the code so that within each time step, it only read from cells and wrote to tmp_cells. Then, after each iteration I simply swapped the pointers between the grids.

C. Arithmetic

Some arithmetic operations are more computationally expensive than others. In particular, divisions involve more steps than operations such as addition or multiplication, since they must perform additional comparisons and can't take advantage of pipelining or parallelism in the same way.

I aimed to minimise the number of division operations using two main methods. The first instance involved common denominators, which I simplified by precalculating the inverse of the denominator, then rewriting the divisions as multiplications with this value.

The second instance involved modulo division, which I replaced with ternary operators. Examining the difference in the Godbolt Compiler Explorer [1] revealed that the number of assembly instructions were halved and latency reduced by over 70% when referencing the computational intensity of each of these instructions for Intel Xeon Broadwell architecture used by BC4 [2].

D. Results

I compiled the original unoptimised code with default compiler settings (gcc v4.8.5 with -O3) as well as the optimised serial code. I tested each build on the different grid sizes under three independent trials, aggregating results with a mean average run time.

TABLE I
UNOPTIMISED VS OPTIMISED SERIAL PERFORMANCE

Grid	U Runtime (s)	O Runtime (s)	Speedup
128x128	33.42	22.26	1.50X
128x256	66.83	44.21	1.51X
256x256	265.58	176.41	1.51X
1024x1024	1,104.69	730.48	1.51X

U (unoptimised), O (optimised).

The results showed a clear 1.5X speedup from the original code, demonstrating that the serial optimisations had successfully improved efficiency.

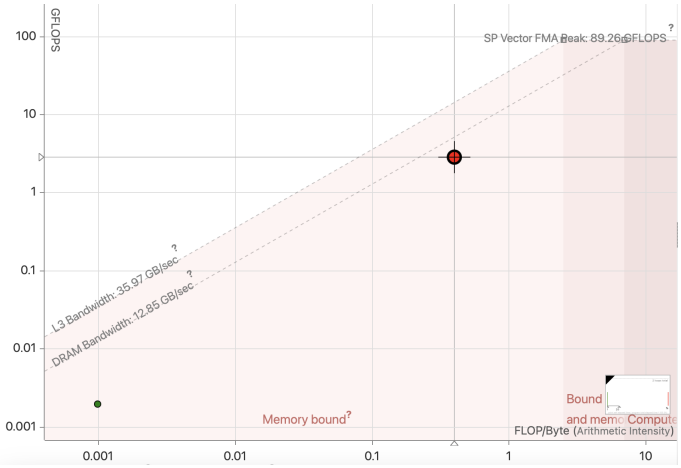

Fig. 2. Roofline graph for the optimised serial code on the 128x128 grid, generated using Intel Advisor. The timestep loop's performance (indicated by the large red dot) was 2.872 GFLOPS, 56% of DRAM peak and 20% of L3 cache peak. The arithmetic intensity was 0.398 FLOPS/byte.

As illustrated in Fig. 2, these optimisations allowed the code to achieve a good proportion (over 50%) of the peak performance possible by a single core. It is clear to see that at this point, the timestep loop was memory bandwidth bound, since the red dot is left of the ridge point.

To improve performance further, I made efforts to increase the operational intensity via vectorisation. Although the same amount of data would need to be loaded, utilisation of memory alignment with SIMD instructions could reduce the number of cache misses and therefore FLOPS per byte transferred.

III. VECTORISATION

A. Setup

The next stage of optimisations was vectorisation with SIMD (single-instruction multiple-data). This is where multiple data items are packed into vectors to be processed simultaneously by the same instruction.

At this stage, my LBM code was structured as follows. Within each time step, the program iterates over every row in the grid. It has a nested loop that iterates over each cell within a row. Thanks to the loop fusion completed in II-B, the cells can be processed completely independently within each time step. I was therefore able to vectorise the inner loop, so that multiple iterations of this loop could be processed in parallel with SIMD.

B. AoS to SoA

In order to prepare the code for vectorisation, I refactored so cells and tmp_cells used a structure of arrays (SoA) instead of array of structures (AoS). This arranged the data such that each speed value would be adjacent in memory, allowing caching to fully exploit data locality. More specifically, adjacent speed values are all loaded on a cache line together. This is ideal for vectorisation, where each of the parallel lanes need to operate on these adjacent data.

C. Compiler Hints

To help the compiler with vectorising the code, I used the restrict keyword on grid pointers so that it would be aware that these pointers would not alias (i.e. refer to the same memory) since vectorisation requires there to be no dependencies between the iterations.

To help the compiler with vectorisation, I also supplied alignment information through the use of the __assume and __assume_aligned directives. Together with _mm_malloc, these ensured that the grid data would always be aligned to 64-byte boundaries, which enables optimal data transfer for the AVX2 architecture used by Broadwell [3].

D. Results

I followed the same process as in II-D to compare the performance of the vectorised code to that of the optimised serial version. The results are shown in table II.

TABLE II
SERIAL VS VECTORISED PERFORMANCE

Grid	S Runtime (s)	V Runtime (s)	Speedup
128x128	22.26	4.96	4.49X
128x256	44.21	14.79	2.99X
256x256	176.41	47.49	3.71X
1024x1024	730.48	233.96	3.12X

S (optimised serial), V (vectorised).

Vectorisation gave a speedup between approximately 3X and 4.5X. It varied between grid sizes, with the smallest 128x128 grid showing greatest increase in performance.

IV. Parallelisation

A. Work-Sharing

To enable work-sharing, I used OpenMP's parallel for directive on as many loops as possible. This prompts the compiler to use the fork-join model on the loop iterations, allocating each iteration to one of the threads. I chose to keep default scheduling options; despite testing various schedule clauses, the static option (which allocates iterations roughly evenly between threads) was the fastest.

Although I did attempt to collapse the outer loop (while continuing to use SIMD), parallelising the outer loop and using SIMD on the inner loop gave better performance.

B. Reduction

Since I moved the av_velocity functionality within the timestep loop, there is a summation over the velocities calculated by each thread. This could lead to race conditions if two threads attempt to write to the shared sum variable at the same time. There are several solutions to this problem, including the use of data sharing clauses such as firstprivate to give each thread its own copy of the sum to work on; or a critical directive, which 'locks' the unsafe region while a thread is accessing it. I chose to use a reduction clause instead, since it is built for this common pattern and provides better optimised results than the alternatives.

C. NUMA-Aware

BC4 nodes have two memory slots, each one near a different socket. A shared-memory approach is used so that every thread has access to all memory, regardless of the socket it runs on. However, if a thread needs to access memory from the slot nearest to the other socket, this data needs to travel via that socket and the interconnect before reaching the thread's core. This is slower than accessing memory from its nearest DRAM slot, giving rise to non-uniform memory access (NUMA).

The operating system allocates memory through a first touch policy. This means that regardless of any malloc calls, data is allocated to the DRAM nearest to the thread that first interacted with it (i.e. initialisation).

To limit the effects of NUMA, I parallelised the initialisation code in an identical way to the main loop. This meant that the data is allocated nearest to the thread that initialises it, which is also the same as the thread that will access it in the main loop, thus reducing the number of fetches from the opposite DRAM.

I also aimed to secure this by setting the OMP_PLACES and OMP_PROC_BIND environment variables so that threads would be pinned close together. This was to prevent the operating system moving the threads across sockets, defeating the purpose of the previous step.

D. Parallel Scaling

To see how the parallel code scaled, I used the OMP_NUM_THREADS environment variable to set the number of threads (and therefore cores) used. I chose to run simulations with even numbers of cores from 2 to 28 (the maximum on a node of BC4). I also ran simulations with a single thread so that I could calculate speedup and parallel efficiency, both of which require this serial time:

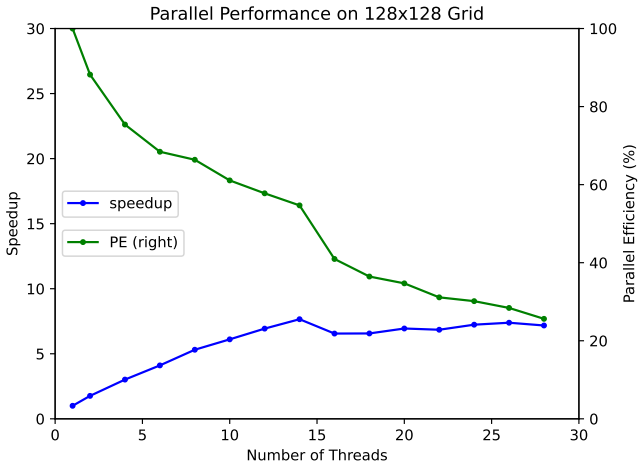

Fig. 3. Parallel scaling for the 128x128 grid.

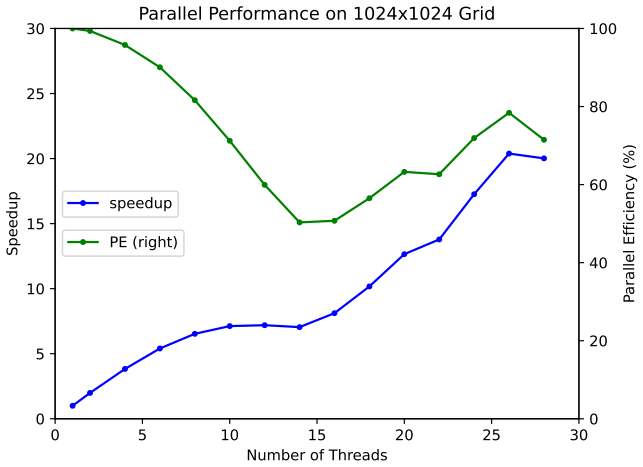

Fig. 4. Parallel scaling for the 1024x1024 grid.

$$\text{Speedup} = \frac{\text{Serial Runtime}}{\text{Parallel Runtime}} \tag{1}$$

$$\text{Parallel Efficiency} = \frac{\text{Speedup}}{\text{Number of Cores}} \tag{2}$$

I plotted results in terms of speedup and parallel efficiency as shown in Fig. 3–4. All calculations were according to strong scaling since grid sizes were kept fixed.

Each grid saw sublinear scaling, and a trend where the parallel efficiency declined with respect to the number of threads.

E. Analysis

In the 128x128 grid, speedup plateaued after 14 threads while parallel efficiency dropped significantly. This suggests that utilising all cores on one socket is optimal for this grid

size. The reason for this is likely that all the grid data is assigned to the memory nearest one socket.

Using a second socket just creates additional overhead in data transfer (since it needs to travel from the further DRAM via the processor interconnect), which balances out any potential benefits of further parallelism.

By contrast, the 1024x1024 grid saw continuous speedup beyond 14 threads, and parallel efficiency actually began to increase. This is likely due to the use of cache associated with the second socket.

BC4 nodes have 35 MiB L3 cache per socket [4]. The 1024x1024 grid takes up at least 76 MiB space in memory (by totalling malloced space for cells , tmp_cells and obstacles).

Due to the settings implemented in IV-C, the first 14 threads are likely assigned to the cores on one socket and all the grid data is allocated to the DRAM slot nearest to this socket.

Since the data is too large to fit in the L3 cache, it has to transfer data to and from DRAM often. However, once we add threads 13–28, we start to make use of the second socket.

Again, thanks to IV-C, these threads are assigned to the second socket and their associated grid segments are allocated to the nearest DRAM slot. Now that the data is split across the two sockets and their corresponding DRAM, it can almost fit entirely within the two L3 caches, eliminating the majority of memory accesses done previously.

F. Final Results

Again, I used the same experiment design as in previous tests, with three independent trials on each grid size. Table III shows a comparison with the vectorised performance and table IV compares the performance between the original unoptimised code and final, fully optimised version. All runtimes quoted are averages across the three trials.

TABLE III
VECTORISED VS PARALLELISED PERFORMANCE

Grid	V Runtime (s)	P Runtime (s)	Speedup
128x128	4.96	0.69	7.19X
128x256	14.79	1.18	12.53X
256x256	47.49	2.84	16.72X
1024x1024	233.96	14.51	16.12X

V (vectorised), P (parallelised).

TABLE IV
UNOPTIMISED SERIAL VS FULLY OPTIMISED PARALLEL PERFORMANCE

Grid	U Runtime (s)	O Runtime (s)	Speedup
128x128	33.42	0.69	48.43X
128x256	66.83	1.18	56.64X
256x256	265.58	2.84	93.51X
1024x1024	1,104.69	14.51	76.13X

U (unoptimised), O (optimised).

The results show massive speedup when compared to both vectorised and serial solutions. Table V compares each runtime to the ballparks (both parallel), demonstrating notably faster times.

TABLE V
BALLPARK VS MY VERSIONS' PERFORMANCE

Grid	B Runtime (s)	M Runtime (s)	Speedup
128x128	0.9	0.69	1.3X
256x256	3.1	2.84	1.1X
1024x1024	19	14.51	1.3X

B (ballpark), M (mine).

Finally, another roofline analysis visualises the improvements made from the optimised serial performance. The timestep loop is no longer bound by DRAM, implying more efficient use of caching. The operational intensity is also 1.3X higher, which means the parallel version performs more operations relative to data transfer. This has shifted the loop closer to the compute bound region, enabling it to access higher performance; in this case a 57X increase in GLOPS from the optimised serial code.

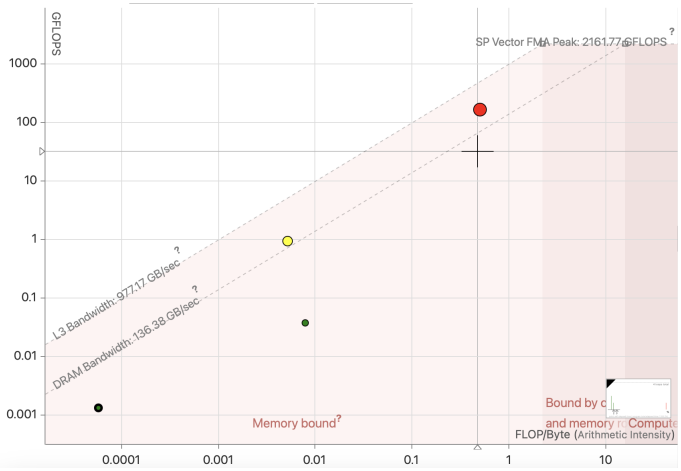

Fig. 5. Roofline graph for the fully optimised parallel code, using all 28 cores on the 128x128 grid, generated using Intel Advisor. The timestep loop's performance (indicated by the large red dot) was 163.399 GFLOPS, 33% of L3 cache peak. The arithmetic intensity was 0.503 FLOPS/byte.

V. CONCLUSION

Each stage generated large performance optimisations both in terms of speedup and resource utilisation. Potential avenues for further optimisation include investigating alternative compiler flags, FMA instructions and more advanced profiling tools to identify other areas for improvement.

REFERENCES

[1] M. Godbolt, "Compiler Explorer," godbolt.org. https://godbolt.org (accessed Mar. 06, 2023).

[2] A. Fog, "Introduction 4. Instruction tables." Available: https://www.agner.org/optimize/instruction_tables.pdf

[3] "Data Alignment to Assist Vectorization," Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html (accessed Mar. 06, 2023).

[4] "Xeon E5-2680 v4 - Intel - WikiChip," en.wikichip.org. https://en.wikichip.org/wiki/intel/xeon_e5/e5-2680_v4