Optimising and Parallelising d2q9-bgk.c

George Herbert

cj19328@bristol.ac.uk

7 March 2022

Abstract

d2q9-bgk.c implements the Lattice Boltzmann method (LBM) to simulate a fluid density on a lattice. This report analyses the techniques I utilised to optimise and parallelise d2q9-bgk.c.

1 Original Code

I ran the provided Makefile to compile the original d2q9-bgk.c, which executed the GNU Compiler Collection (GCC) (version 4.8.5) with the -std=c99 -O3 and -lm options. Table 1 contains the execution time for each test case.

Table 1: Execution times of the original code

Grid Size	Time (s)
128×128	29.16
128×256	58.71
256×256	233.32
1024×1024	980.89

Each time was an average of five runs on a BlueCrystal Phase 4 (BC4) compute node—a Lenovo nx360 M5, which contained two 14-core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs and 128 GiB of RAM [1]. I took an average because of the variation between runs, which existed due to the inconsistent performance of compute nodes.

2 Compiler Optimisations

2.1 Intel C Compiler Classic

I hypothesised that compiling with the Intel C Compiler Classic (ICC) instead of GCC would produce an executable better optimised for BC4's Intel compute nodes. Table 2 contains the execution times after compiling with ICC (version 19.1.3.304). Primarily, the speedup was because ICC vectorized two of the innermost nested loops in the collision procedure, which GCC did not.

Table 2: Execution times using ICC and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	22.54	1.29
128×256	45.56	1.29
256×256	179.06	1.30
1024×1024	805.21	1.22

2.2 Compiler Options

I hypothesised that I could compile my program with more aggressive optimisations to improve performance whilst still producing the correct solution. I compiled my code with the -Ofast option, which implemented additional aggressive floating-point optimisations [2]. These aggressive optimisations provided a small performance boost, as evident in Table 3.

Table 3: Execution times using new ICC options and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	22.18	1.02
128×256	44.42	1.03
256×256	176.69	1.01
1024×1024	794.44	1.01

3 Serial Optimisations

I used the Intel Advisor tool to generate a Roofline model, as shown in Figure 1.
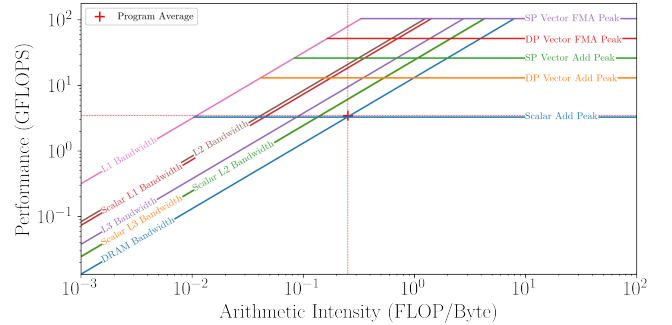

Figure 1: Roofline model following compilation changes, run on the 1024×1024 test case

I identified several attributes of my program and several key areas to focus my serial optimisations. For larger grid sizes especially, the LBM implementation in d2q9-bgk.c was a memory bandwidth bound problem. d2q9-bgk.c achieved an average arithmetic intensity of 0.25 FLOP/byte and performance of 3.44 GFLOPS for the 1024×1024 test case.

3.1 Loop Fusion

Using the Roofline model, I identified that decreasing the number of accesses to memory within each timestep would simultaneously increase my program's performance whilst also increasing the arithmetic intensity, thus

raising the performance bound for my latter implementations.

To accomplish this, I implemented loop fusion. In the original code, the entire grid was iterated over in four sequential procedures within each timestep: `propagate`, `rebound`, `collision` and `av_velocity`. I fused the four loops within these procedures into a single loop in the `timestep` function, which enabled me to eliminate the temporary allocations of cell values to the `tmp_cells` array. Instead, my program wrote new values of cells to the `cells_new` array; each timestep, my program swapped the pointers of `cells` and `cells_new`.

Table 4 displays the improvements to the execution time. Moreover, my program's arithmetic intensity increased to 0.29 FLOP/byte, and the performance increased to 3.54 GFLOPS for the 1024×1024 test case.

Table 4: Execution times with loop fusion and pointer swap, and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	19.42	1.15
128×256	39.21	1.13
256×256	155.64	1.14
1024×1024	635.61	1.25

3.2 Arithmetic

Despite the compiler optimising the arithmetic within each timestep, I hypothesised I could make some manual improvements to reduce my program's execution time further.

Division operations take considerably more time to execute than other basic arithmetic operations, such as multiplication. I precalculated several values to eliminate unnecessary division operations: `c_sq_r`, the reciprocal of c^2; `two_c_sq_r`, the reciprocal of $2c^2$; and `two_c_sq_sq_r`, the reciprocal of $2c^4$, where c was the speed of sound. I then replaced several division operations with multiplications by these values.

Additionally, I eliminated the repeated calculation of and division by `tot_cells` (i.e. the number of cells that were not obstacles) each timestep. Instead, my program calculated `tot_cells` once during the initialisation phase. It then stored the reciprocal of this value in `params.num_non_obstacles_r`, which was multiplied with `tot_u` to compute each timestep's average velocity.

These arithmetic improvements provided only a slight speedup compared to the previous implementation, as shown in Table 5; this was because memory accesses dominated my program's execution time. I would expect a more considerable speedup in smaller grid sizes than the test cases provided.

4 Vectorization

Vectorization is the process of converting a scalar implementation to a vector implementation, which enables the compiler to use additional registers to perform single, multiple data (SIMD) operations.

Table 5: Execution times with arithmetic improvements and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	19.10	1.02
128×256	38.49	1.02
256×256	153.39	1.01
1024×1024	621.52	1.02

4.1 Structure of Arrays

I hypothesised that converting the cells' variables from an array of structures (AoS) to a structure of arrays (SoA) format would suit vectorisation of the inner loop; the SoA format would keep memory accesses contiguous over structure instances. I altered the `t_speed` structure to contain nine pointers, each to an individual array of floats. Each array of floats contained the values of one vector for each cell within the grid.

The SoA format increased the execution time of my program, as shown in Table 6. Upon inspection, the AoS format had better memory locality in the program's primarily unvectorized form.

Table 6: Execution times with SoA and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	19.25	0.99
128×256	51.25	0.75
256×256	211.45	0.73
1024×1024	1102.18	0.56

4.2 Memory for Vectorization

To effect vectorization of the inner loop, I identified that I needed to align the `cells`, `cells_new` and `obstacles` variables and hint to the compiler that the arrays in the `t_speed` structure were not aliased.

Processors efficiently move data located on specific byte boundaries, and compilers can perform optimisations when data access is known to be aligned by 64 bytes [3]. To align the `cells`, `cells_new` and `obstacles` variables, I replaced calls to the `malloc` and `free` procedures with the alignment specific replacements: `_mm_malloc` and `_mm_free`, respectively. I used the `__assume__aligned` procedure and the statement `__assume(params.nx % 16 == 0)` to inform the compiler that the dynamically allocated variables were aligned. Doing so prevented the compiler from generating conservative code, which would have been detrimental to the speed of my implementation.

I compiled my code with the `-restrict` option and used the `restrict` keyword to define each of the nine pointers in `t_speed`. The `restrict` keyword asserted that these pointers were not aliased, which prevented the compiler from performing a runtime check for aliasing.

I implemented the `#pragma omp simd` pragma to direct the compiler to utilise SIMD instructions to execute operations within the inner loop in `timestep`. I used the

`reduction(+:tot_u)` clause to ensure the `tot_u` variable contained the correct value at the loop's termination.

I compiled with the `-xAVX2` option to tell the compiler to optimise for Intel processors that support Advanced Vector Extensions 2 (AVX2) (which BC4's compute nodes do) [4]. I did not compile explicitly with the `-qopenmp-simd` option since `-Ofast` enabled it by default [2].

Vectorization provided the most considerable improvement of any optimisation that I had implemented to this point, as shown in Table 7. The compiler could produce AVX2 instructions that could perform simultaneous operations on up to eight single-precision floating-point numbers. Furthermore, my implementation achieved an arithmetic intensity of 0.43 FLOP/byte and performance of 10.14 GFLOPS when run on the 1024×1024 test case.

Table 7: Execution times after vectorizing and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	5.77	3.34
128×256	11.57	4.43
256×256	41.55	5.09
1024×1024	215.52	5.11

5 Parallelism

5.1 OpenMP

Using the Intel Advisor tool, I identified that a large proportion of my program's execution time was in the inner loop in the `timestep` function. Therefore, I hypothesised that parallelising the outer loop would substantially improve performance.

OpenMP implements parallelism by launching a set of threads that execute portions of code concurrently. I included OpenMP's `#pragma omp parallel for` pragma, and compiled my code with the `-qopenmp` option to parallelise the outer loop. I used the clause `reduction(+:tot_u)` to prevent race conditions; the reduction clause informed the compiler to create a copy of the `tot_u` variable for each thread (initialised to zero) and to sum the local results at the end of the parallel region. I used a static schedule to limit the variance between runs.

Table 8 displays the execution times for my parallel implementation when run with 28 threads. There was a significant speedup over the vectorised implementation because the program could execute iterations of the inner loop in parallel and utilise more cache memory.

5.2 Non-Uniform Memory Access

Non-uniform memory access (NUMA) is a computer memory design in which memory access time depends on the memory location relative to the processor. I identified that memory for the `cells` and `obstacles` variables

Table 8: Execution times with 28 threads and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	1.14	5.06
128×256	1.35	8.57
256×256	3.33	12.48
1024×1024	14.38	14.99

was allocated the first thread in my prior implementation, per the first touch policy. Therefore, I hypothesised that a NUMA-aware implementation would reduce the average memory access time.

I parallelised the initialisation loops for `cells` and `obstacles` to ensure that each thread touched the same data in the `initialise` and `timestep` procedures. Additionally, I set the environment variables `OMP_PROC_BIND=true` and `OMP_PLACES=cores` to prevent threads from moving cores.

Table 9 contains the updated execution times for my final NUMA-aware implementation. The NUMA-aware implementation produced a reasonable performance boost by reducing the average time to access a memory location.

Table 9: Execution times with 28 thread NUMA-aware implementation and speedup over both the prior and vectorized implementation

Grid Size	Time (s)	Speedup	
		Prior	Vectorized
128×128	0.72	1.58	8.01
128×256	0.80	1.69	14.46
256×256	2.47	1.35	16.82
1024×1024	12.81	1.23	16.83

5.3 Scaling

I ran my final implementation from 1–28 threads to analyse how my program scaled. I then calculated the subsequent threads' speedup over a single thread implementation. Figure 2 displays the resultant speedup curves.

In general, my implementation initially scaled well for each grid size, but the speedup acquired from each subsequent thread declined (i.e. a sublinear plateau), which occurred due to several reasons. Firstly, perfect linear scaling was theoretically impossible because the program contained serial sections. Secondly, threads had to interact with cells stored in different NUMA regions in the parallel implementation. Lastly, threads incur an overhead; for example, when summing the local copies of `tot_u`.

Notably, the amount of speedup provided by each next core was approximately inversely proportional to the test case size. In other words, larger grid sizes benefitted more from a multithreaded implementation than smaller grid sizes. Firstly, this was because the larger grids benefitted more from being split sufficiently small to fit into the faster cache levels. Secondly, the larger grid sizes
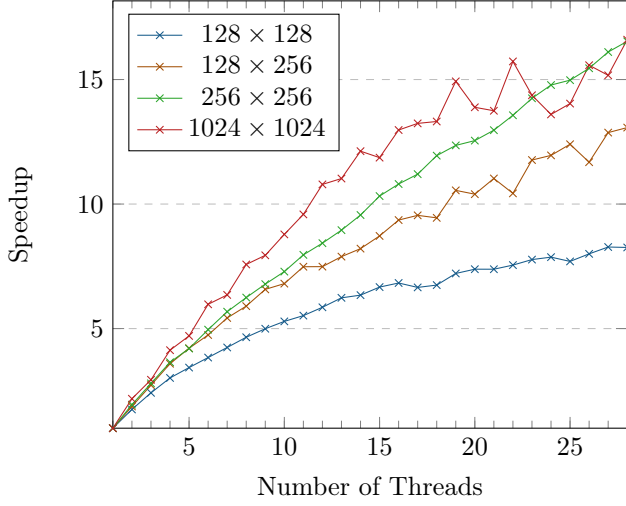
Figure 2: Speedup curves for my NUMA-aware implementation

were more evenly divided by the number of threads.

5.4 Comparison to Serial

I used the Intel Advisor tool to analyse the performance of my final implementation, as shown in Figure 3. On the 1024×1024 test case, my program achieved an arithmetic intensity of 0.43 FLOP/byte and 168.35 GFLOPS of performance.
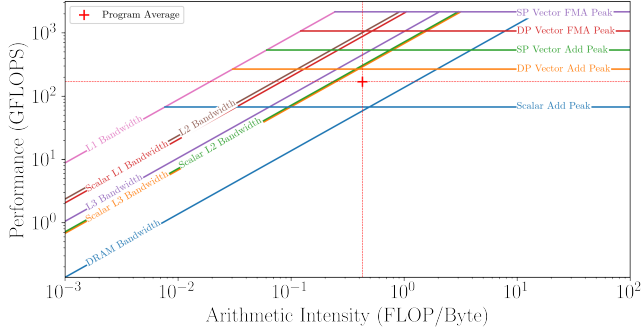

Figure 3: Roofline model of my final implementation, run on the 1024×1024 test case

Compared to my vectorized implementation, the arithmetic intensity was identical, whereas the performance increased by a factor of 16.60. The increase in performance was primarily due to two main reasons. Firstly, by running across multiple threads on multiple cores, the processor could compute sections of the grid in parallel. Secondly, the parallel program did not have to interact with the DRAM, which had a lower bandwidth, as often. When running on 28 cores, the parallel program had twice as much L3 cache, and 28 times more L1 and L2 cache than the serial program. For example, in the `timestep` loop in the vectorized implementation, 5130.56 GB and 2306.03 GB of data were passed through the L1 cache and DRAM, respectively. In the NUMA-aware implementation, 5777.32 GB and 1447.21 GB of data were passed through the L1 cache and DRAM, respectively.

6 LBM in Go

Unlike C, Go is a language with concurrency built into its core design; consequently, one can implement parallelism effortlessly. I sought to investigate whether one could achieve a similar level of performance using Go's core features. I deemed this important since people frequently sacrifice small amounts of performance for a more straightforward implementation in real-world scenarios.

I produced an implementation of LBM in Go, `d2q9-bgk.go`, using an almost identical algorithm to that in `d2q9-bgk.c`. The Go implementation was significantly slower in experiments, as evident in Table 10.

Table 10: Execution times of `d2q9-bgk.go`, and speedup compared to the final `d2q9-bgk.c`

Grid Size	Time (s)	Speedup
128×128	7.66	0.09
128×256	13.10	0.06
256×256	41.83	0.06
1024×1024	103.02	0.12

The reasons for the comparatively poor performance included poor compiler optimisations and an unvectorized inner loop. Methods exist to rectify these problems partially; however, they are directly contradictory to the desired simplicity of the experiment. In conclusion, one cannot achieve a comparable level of performance from a more straightforward implementation using Go's core features.

7 Conclusion

In conclusion, the optimisations I utilised significantly improved the performance of `d2q9-bgk.c`. Across the increasingly large test cases, my program achieved a speedup of 40.5x, 73.39x, 96.46x and 76.57x compared to the original implementation. Vectorization and OpenMP multithreading provided the most significant improvements because both implemented some degree of parallelism: data level and task level, respectively.

References

[1] *BlueCrystal technical specifications*. URL: https://www.bristol.ac.uk/acrc/high-performance-computing/hpc-systems-tech-specs/ (visited on 19/02/2022).

[2] *Alphabetical List of Compiler Options*. 12th June 2021. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/alphabetical-list-of-compiler-options.html (visited on 20/02/2022).

[3] *Data Alignment to Assist Vectorization*. 22nd Jan. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html (visited on 22/02/2022).

[4] *Lenovo NeXtScale nx360 M5 (E5-2600 v4)*. URL: https://lenovopress.com/lp0094-nextscale-nx360-m5-e5-2600-v4 (visited on 22/02/2022).