Optimising and Parallelising d2q9-bgk.c

George Herbert
cj19328@bristol.ac.uk

March 3, 2022

Abstract

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

1 Original Code

I ran the provided `Makefile` to compile the original `d2q9-bgk.c` code, which executed the GNU Compiler Collection (GCC) 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

I measured the original code to quantify the performance improvements of my latter implementations. Each time is an average of five runs on BlueCrystal Phase 4's (BC4's) compute nodes; each of which was 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.

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].

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

Grid Size	Time (s)	Speedup
128 × 128		
128 × 256		
256 × 256		
1024 × 1024		

These changes to the compilation process provided a good 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	22.25	
128 × 256	44.42	
256 × 256	176.69	
1024 × 1024	795.41	

3 Serial Optimisations

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

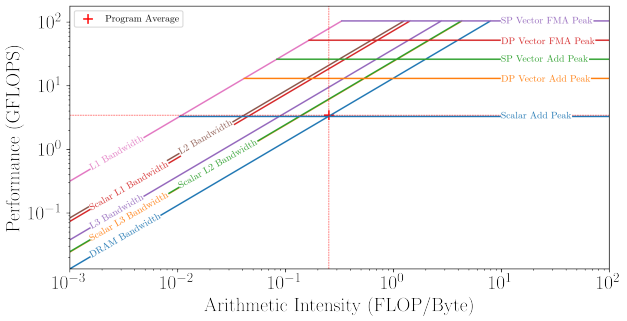

Figure 1: Roofline chart following compilation changes, run on the 1024 × 1024 test case

I identified several attributes of my program and several key areas to focus my 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 chart, 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. This enabled me to eliminate the temporary allocations of cell values to the `tmp_cells` array. Instead, all new values of cells are written to the `cells_new` array, and the pointers of `cells` and `cells_new` are swapped each timestep.

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 further reduce my program's execution time.

Division operations take considerably more time to execute than other basic arithmetic operations, such as multiplication. To eliminate a several unnecessary division operations, I precalculated several values: `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 is the speed of sound. I then replaced several division operations with multiplications by these values.

Additionally, I eliminated the repeated calculation and division of `tot_cells` (i.e. the number of cells that were not obstacles) each timestep. To do so, my program calculated `tot_cells` once during the initialisation phase. I then set the reciprocal of this value in the `params.num_non_obstacles_r` variable, which I multiplied with `tot_u` to compute the average velocity each timestep.

These arithmetic improvements provided only a slight speedup compared to the prior implementation, as shown in Table 5. This was because the execution time of my program was dominated by memory accesses. I would expect a larger speedup in smaller grid sizes than the test cases provided.

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 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.

4.1 Structure of Arrays

I hypothesised that converting the `t_speed` structrue from an array of structures (AoS) to a structure of arrays (SoA) would suit vectorisation of the inner loop; the SoA format would keep memory accesses contiguous over structure instances. I altered the `t_speed` 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 actually increased the execution time of my program, as shown in Table 6. This is because the AoS format has better memory locality in the program's unvectorized form.

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

Grid Size	Time (s)	Speedup
128×128		
128×256		
256×256		
1024×1024		

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 gener-

ating 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 the memory referenced by these pointers was not aliased. This prevented the compiler from performing a runtime check for aliasing.

I implemented the `#pragma omp simd` pragma, indicated to the compiler to utilise SIMD instructions to execute operations within the inner loop. I utilised 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 direct the compiler to optimise for Intel processors that support Advanced Vector Extensions 2 (AVX2) (which BC4's compute nodes do) [4]. I did not have to compile with any additional options to enable the compiler to follow OpenMP SIMD directives since the `-qopenmp-simd` option was enabled by default at the `-Ofast` optimisation level [2].

Vectorization provided the most considerable improvement to speedup of any optimisation that I had implemented to this point, as shown in Table 7. This is because 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 with vectorization and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	5.77	
128×256	11.57	
256×256	41.55	
1024×1024	215.52	

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 provide a substantial preformance improvement.

OpenMP implements parallelism by launching a set of threads that execute portions of code concurrently. To parallelise the outer loop, I included OpenMP's `#pragma omp parallel for` pragma, and compiled my code with the `-qopenmp` option. 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 when the outer loop terminated.

Table 8 displays the execution times for my parallel implementation, when run with 28 threads. There is a significant speedup over the vectorized implementation because the program is able to utilise a larger amount of L1 and L2 cache, and execute iterations of the inner loop in parallel.

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

Grid Size	Time (s)	Speedup
128×128	1.14	
128×256	1.35	
256×256	3.33	
1024×1024	14.38	

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 were being allocated the first thread in my prior implementation, in accordance with 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 both 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 virtue of speeding up the average time to access a memory location.

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

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

5.3 Scaling

I ran my final implementation from 1–28 threads to analyse how my program scaled. I then calculated the speedup that subsequent threads provided over a single

thread implementation. Figure 2 displays the resultant speedup curves.

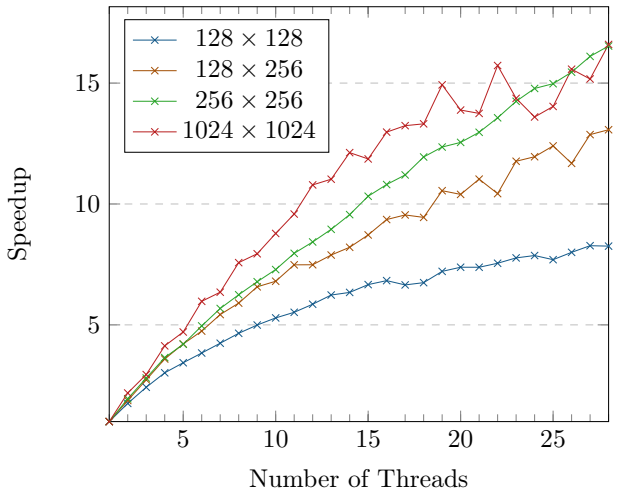

Figure 2: Speedup curves for my NUMA-aware implementation

In general, my implementation initially scaled well for each grid size, but the speedup acquired from each subsequent thread declined—this is known as a sublinear plateau. Notably, the amount of speedup provided by each subsequent core was approximately inversely proportional to the test case size. In other words, larger grid sizes benefit more from a multithreaded implementation than smaller grid sizes—this is because the smaller grids saturate the memory bandwidth sooner.

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.

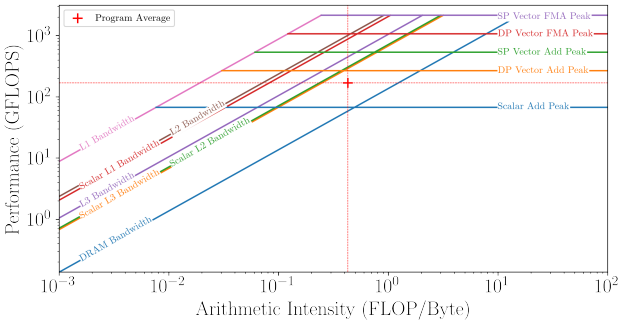

Figure 3: Roofline chart of final implementation, run on the 1024×1024 test case

Compared to my vectorized implementation, the arithmetic intensity was identical, whereas the performance increased by a factor of 16.60. This was primarily due to two reasons. Firstly, by running across multiple threads on multiple cores, sections of the grid could be computed in parallel. Secondly, inspecting the Intel Advisor memory metrics for the 1024×1024 grid size highlighted that the parallel program did not have to interact with the DRAM, which had a lower bandwidth, as often. 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. Whereas 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

Many other languages, APIs, and libraries can be utilized to implement LBM; all have associated advantages and disadvantages.

Unlike C, Go is a language with concurrency built into its core design. Consequently, one can achieve parallelism in Go effortlessly, without requiring significant knowledge of the underlying system or APIs such as OpenMP. As a final, interesting experiment, I produced an implementation of LBM in Go to identify whether Go's easily-understood core features could achieve a comparable amount of performance.

In experiments, the Go implementation was between seven and sixteen times slower. Specifically, it achieved a time of 7.66 seconds, 13.10 seconds, 41.83 seconds and 103.02 seconds on the increasingly large test cases. Upon inspection, there were likely several reasons for the comparatively poor performance. Firstly, Go's compiler did not perform many optimisations. Compiling with GCCGO or GOLLVM may have produced a better-optimised executable; however, these required a complicated installation process, which was directly contradictory to the desired simplicity for the experiment. Secondly, and perhaps most prominently, was the lack of vectorization. Vectorizing code in Go requires external libraries or assembly, which similarly conflicted with the desired simplicity.

In conclusion, a similar implementation built with Go's easily-understood core features could not achieve an even remotely comparable level of performance.

7 Conclusion

References

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

[2] *Alphabetical List of Compiler Options*. June 12, 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 Feb. 20, 2022).

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

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