Optimising and Parallelising d2q9-bgk.c

George Herbert
cj19328@bristol.ac.uk

February 28, 2022

Abstract

d2q9-bgk.c implements the Lattice Boltzmann methods (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 following command:

```
gcc -std=c99 -Wall -O3 d2q9-bgk.c
    -lm -o d2q9-bgk.
```

Table 1: Execution times of the original code

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

Table 1 contains the total time to initialise, compute and collate each test case when running the ELF file produced. It was essential to measure the original code to quantify the performance improvements of my latter implementations. I measured each of the total times by taking an average of 10 runs on BlueCrystal Phase 4's (BC4's) compute nodes. Each of BC4's compute nodes 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 of multiple runs because of the variation between runs, which existed due to the inconsistent performance of compute nodes.

2 Serial Optimisations

2.1 Compiler

The first improvement I implemented was compiling with the Intel C Compiler instead of GCC since it produced machine code better optimised for BC4's Intel compute nodes. Furthermore, I compiled my code with the Ofast option, which set aggressive options to improve the speed of my program, including O3 optimisations and aggressive floating-point optimisations [2].

Table 2: Execution times after compilation changes, and speedup over the original code

Grid Size	Time (s)	Speedup
128 × 128	22.25	1.31
128 × 256	44.42	1.32
256 × 256	176.69	1.33
1024 × 1024	795.41	1.23

These changes to the compilation process provided a good performance boost, as shown in Table 2.

2.2 Loop Fusion and Pointer Swap

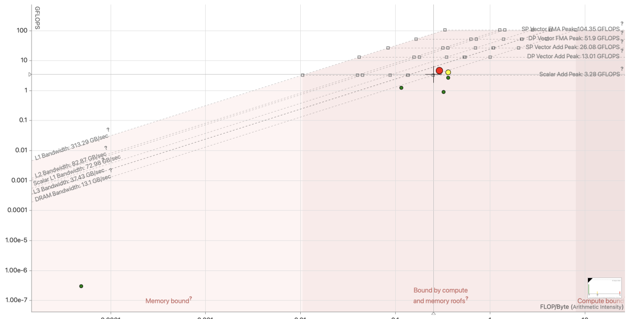

Figure 1: Screenshot of the Intel Advisor Roofline Analysis for my implementation following compilation changes, run on the 1024 × 1024 test case

Once I had improved the compilation process, I used the Intel Advisor tool to generate a Roofline chart, as shown in Figure 1. I identified several attributes of my program and several key areas to focus my optimisations. Firstly, 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.

From this, I identified a significant opportunity to optimise `d2q9-bgk.c` by decreasing the number of memory accesses. Decreasing the quantity of bytes accessed would simultaneously increase the performance of the implementation whilst also increasing the arithmetic intensity, which would have the additional benefit of increasing the performance bound for my latter implementations.

One method I utilised to accomplish this was 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`. By absorbing these four procedures into the `timestep` procedure and fusing the four loops, I drastically decreased the number of memory accesses, thereby improving the performance of my program.

Implementing loop fusion offered another significant opportunity to eliminate redundant memory accesses. The original code contained substantial value copying between the `cells` and `tmp_cells` arrays. I was able to eliminate this by writing all new values of cells to a `cells_new` array, and simply swapping the pointers of `cells_new` and `cells` at the end of each timestep. I eliminated the `tmp_cells` array.

Table 3: Execution times after loop fusion and pointer swap, and speedup over the original code

Test Case Size	Time (s)	Speedup
128×128	19.42	1.50
128×256	39.21	1.50
256×256	155.64	1.50
1024×1024	635.61	1.54

Table 3 displays the overall performance improvements. Furthermore, as a result of these optimisations, the arithmetic intensity increased to 0.29 FLOP/Byte and the performance increased to 3.54 GFLOPs for the 1024×1024 test case.

2.3 Arithmetic Improvements

Despite the compiler being able to partially optimise the arithmetic within each timestep without making any changes to the code, there were still some manual improvements that I made to improve the program's performance. Division operations take considerably more time to execute than other basic arithmetic operations, such as multiplication Therefore, to eliminate a large number of unnecessary division operations, I precalculated several values, including:

$$\frac{1}{c^2} = 3 \qquad \frac{1}{2c^2} = 1.5 \qquad \frac{1}{2c^4} = 4.5$$

where c is the speed of sound. Additionally, I noticed that the number of cells in the grid that were not obstacles `tot_u` was recalculated and then divided by each timestep. I eliminated this inefficiency by counting the number of cells that were not obstacles only once (during the initialisation phase). I then saved the reciprocal of this value as a parameter `num_non_obstacles_r`, which I used once per timestep in a multiplicative operation to compute the average velocity.

Table 4: Execution times after arithmetic improvements, and speedup over the original code

Test Case Size	Time (s)	Speedup
128×128	19.10	1.53
128×256	38.49	1.53
256×256	153.39	1.52
1024×1024	621.52	1.58

These arithmetic improvements provided only a slight boost to performance compared to the prior implementation, as shown in Table 4. However, this was unsurprising since the Intel C Compiler had already optimised a large amount of the arithmetic.

2.4 Vectorization

Vectorization is the process of converting a scalar implementation to a vector implementation, which enables the compiler to use additional registers to perform multiple operations in a single instruction [3]. I utilised several techniques to enforce single-instruction-multiple-data (SIMD) vectorization of the inner loop within each timestep.

Firstly, I converted the `t_speed` structure holding cell speeds from an array of structures (AoS) to a structure of arrays (SoA). Previously, an array of t_speed structures represented the grid, whereby each structure contained nine vectors (represented by an array of nine floats) I altered this such that one t_speed structure containing nine pointers, each to an individual array of floats, represented the grid. Each array of floats contained the values of one vector for each cell within the grid. The SoA format greatly suited vectorisation of the inner loop since it kept memory accesses contiguous over structure instances [4].

Having altered the data layout to suit vectorization, I utilised several other techniques to enforce vectorization. I implemented the `#pragma omp simd` pragma to vectorise the inner loop within each timestep. This pragma indicated to the compiler to utilise SIMD instructions to execute operations within the inner loop on multiple data elements in a single instruction. Since I was compiling my code with the `Ofast` optimisation level, the `qopenmp-simd` option was already enabled to utilise OpenMP SIMD compilation [2]. Furthermore, I utilised the `reduction(+:tot_u)` clause to ensure the `tot_u` variable contained the correct value at the loop's termination.

I compiled my code with the `restrict` option and used the `restrict` keyword in `timestep`'s parameters for the `cells`, `cells_new` and `obstacles` variables. The `restrict` keyword asserted that the memory referenced by pointers to these variables was not aliased. Overall, this provided a performance advantage as it prevented the compiler from performing a runtime test for aliasing.

Processors efficiently move data located on specific byte boundaries by the nature of their design, and compilers can perform optimisations when data access is known to be aligned by 64 bytes [5]. 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 performance.

Once I had utilised these techniques to enforce efficient vectorization of the inner loop, I compiled `d2q9-bgk.c` with the `xAVX2` option to direct the compiler to optimise for Intel processors that support Advanced Vector Extensions 2 (AVX2), as BC4's compute nodes do [6].

Table 5: Execution times after vectorization, and speedup over the original code

Test Case Size	Time (s)	Speedup
128×128	5.77	5.05
128×256	11.57	5.07
256×256	41.55	5.62
1024×1024	215.52	4.55

Vectorization provided the most considerable performance improvement of any optimisation that I had implemented to this point, as shown in Table 5. This significant improvement was unsurprising since AVX2 could perform simultaneous operations on up to eight single-precision floating-point numbers.

3 Parallelism

3.1 OpenMP

OpenMP implements parallelism by launching a set of threads that execute portions of code concurrently [7]. I utilised OpenMP's `#pragma omp parallel for` pragma to direct the compiler to parallelise the outer loop in the `timestep` procedure. Furthermore, I compiled my code with the `qopenmp` option, which enabled the parallelizer to generate multithreaded code based on OpenMP directives, as that which I defined. Since the `tot_u` variable needed to contain the total velocities of each cell, 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 terminates.

Table 6: Execution times after parallelising (run with 28 threads), and speedup over both the original and vectorized code

		Speedup	
Grid Size	Time (s)	Original	Vectorized
128×128	1.30	22.43	4.44
128×256	1.48	39.67	7.82
256×256	3.61	64.63	11.51
1024×1024	16.93	57.94	12.73

Table 6 displays the execution times for my parallel implementation (run with 28 threads), and speedup over both the original and vectorized code.

3.2 Non-Uniform Memory Access (NUMA)

NUMA is a computer memory design in which memory access time depends on the memory location relative to the processor [8]. Memory is allocated to the closest NUMA region to the thread that first touches the data [9]. Since BC4's compute nodes contain two sockets, the memory access time for a given thread primarily depends on whether the memory is connected to the socket the thread resides in or not. I parallelised the initialisation loops for `cells` and `obstacles` to ensure that each thread touched the same data in both the `initialise` and `compute` procedures. Furthermore, I set the environment variables `OMP_PROC_BIND=true` and `OMP_PLACES=cores` to prevent threads from moving cores.

Table 7: Execution times after writing NUMA-aware implementation (run with 28 threads), and speedup over both the original and vectorized code

Grid Size	Time (s)	Speedup	
		Original	Vectorized
128×128	0.71	41.07	8.13
128×256	0.82	71.60	14.11
256×256	2.64	88.38	15.73
1024×1024	13.47	72.82	16.00

Table 7 contains the updated execution times for my final NUMA-aware implementation. As anticipated, parallelising the initialisation loops and preventing threads from moving cores provided a modest boost to performance.

3.3 Scaling

I ran my final, NUMA-aware implementation from one to 28 threads to gain an insight into how my implementation scaled.

I calculated the speedup that subsequent threads provided over a single thread implementation. Figure 2 displays the resultunt speedup curves. In general, my implementation initially scaled well for each grid size, but the speedup acquired from each subsequent declines—this is known as a sublinear plateau. There are some exceptions to this trend. Most evidently, there is a significant jump in speedup for the 128×256 test case at 27 cores. Whilst this could be due to noise,

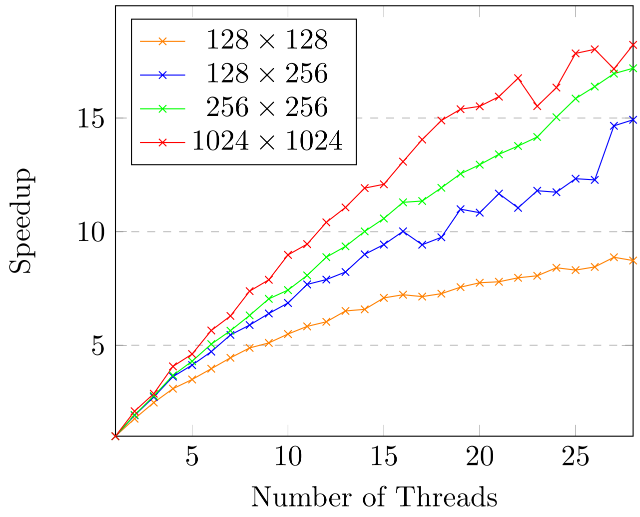

Figure 2: Speedup curves for my NUMA-aware implementation

it is also potentially due to the program splitting the grid sufficiently small to fit on each core's L1 or L2 cache.

Notably, the amount of speedup provided by each subsequent core is inversely proportional to the test case size. In other words, larger grid sizes benefit more from a multithreaded implementation than smaller grid sizes.

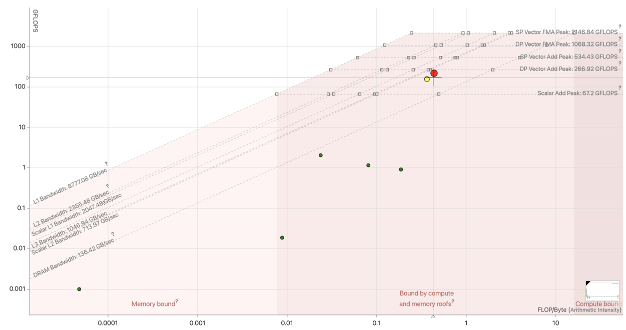

Figure 3: Screenshot of the Intel Advisor Roofline Analysis for my NUMA-aware implementation, run with 28 threads on the 1024×1024 test case

Figure 3 displays the Roofline chart of my final implementation.

4 Alternate implementations

There are many other languages, APIs and libraries that can be utilised to implement LBM, each with their own advantages and disadvantages.

I produced a separate implementation of LBM in Go.

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] *Using Automatic Vectorization.* June 12, 2021. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/vectorization/automatic-vectorization/using-automatic-vectorization.html (visited on Feb. 21, 2022).

[4] *Memory Layout Transformations.* Mar. 26, 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html (visited on Feb. 22, 2022).

[5] *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).

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

[7] *OpenMP: How it Works.* URL: https://cvw.cac.cornell.edu/openmp/threads.

[8] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors.* 3rd. Newnes.

[9] Tom Deakin. “OpenMP and Non-Uniform Memory Access (NUMA)”.