Optimisations and Parallelism of d2q9-bgk.c

George Herbert
`cj19328@bristol.ac.uk`

February 21, 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`, as well as a detailed analysis of those techniques. To do so, this report is split into several sections corresponding to different iterations of my code.

1 Original code

I compiled the original `d2q9-bgk.c` using 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

Figure 1 contains the total time to initialise, compute and collate each of the test cases when running the ELF file produced. It was important to measure the original code, so that I could 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 is a Lenovo nx360 M5, which contains 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 exists due to the inconsistent performance of compute nodes.

2 Serial Optimisations

2.1 Compiler

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

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

I compiled my serial optimised implementation using the Intel® C Compiler as opposed to GCC, since it provides better optimised code for Intel processors. Furthermore, I compiled my code with the `Ofast` flag, which set aggressive options to improve the speed of my program, including `O3` optimisations and aggressive floating point optimisations [2].

2.2 Loop fusion and pointer swap

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

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

LBM is a memory bound problem. As a result of this, there was a significant opportunity to optimise `d2q9-bgk.c` by decreasing the number of memory accesses. 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 fusing the four loops in these procedures into one, I was able to drastically decrease 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 had a significant quantity of 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. Furthermore, I eliminated the `tmp_cells` array entirely.

2.3 Arithmetic improvements

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

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

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 performance of the program. Division is a very slow arithmetic operation relative to 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 number of cells that were not obstacles only once (during the initialisation phase). I then saved the reciprocol 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 5: Execution times after vectorization, and speedup over the original code

Test Case Size	Time (s)	Speedup
128×128	5.79	
128×256	11.66	
256×256	40.81	
1024×1024	213.53	

2.4 Vectorization

3 Parallelised

3.1 OpenMP

3.2 Results

Table 6: Execution times after parallelising, and speedup over both the original and vectorized code

		Speedup	
Grid Size	Time (s)	Original	Vectorized
128×128	0.79		
128×256	0.91		
256×256	2.85		
1024×1024	13.43		

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).