Optimisations and Parallelism of d2q9-bgk.c

George Herbert
cj19328@bristol.ac.uk

February 20, 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 d2q9-bgk.c -lm -o
    d2q9-bgk.
```

Table 1: Total time of the original code for test cases of different sizes

Test Case Size	Time (s)
128×128	0
128×256	0
256×256	0
1024×1024	0

Figure 1 contains the total time to initialise, compute and collate each of the test cases when running the ELF file. 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 Changes

I changed from GCC to the Intel® C Compiler, since it usually 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].

I applied loop fusion to decrease the number of memory accesses in my code. Previously, 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.

Moreover, as a result of implementing the loop fusion, I was also able to implement a pointer swap to eliminate unnecessary swaping between the cells and tmp_cells arrays. New values of cells are written to the cells_new array (renamed from tmp_cells to more accurately describe its new purpose). At the end of each timestep, the memory locations the cells_new and cells variables are pointing to are swapped.

I also improved the arithmetic within each timestep to improve the performance.

2.2 Results

Table 2: Total time of the serial optimised code for test cases of different sizes

Test Case Size	Time (s)
128×128	0
128×256	0
256×256	0
1024×1024	0

3 Vectorization

3.1 Changes

3.2 Results

Table 3: Total time of the vectorized code for test cases of different sizes

Test Case Size	Time (s)
128×128	0
128×256	0
256×256	0
1024×1024	0

4 Parallelism

4.1 OpenMP

4.2 Results

Table 4: Total time of the parallelised code for test cases of different sizes

Test Case Size	Time (s)
128×128	0
128×256	0
256×256	0
1024×1024	0

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).