HPC Lattice Boltzmann: Parallelisation with OpenMP

Alice Ferreira
1913684

1 Serial Implementation

1.1 Initial Runtimes

Initially, the average unoptimised serial runtime of the program was tested for each given input. The program was compiled using GCC with no optimisation flags, and the average runtime on each input size measured. The results are found in Table 1. For the purposes of this report, all average runtimes noted have been measured by executing the same job three times and calculating the mean runtime.

Table 1: Unoptimised average runtimes for each input size in seconds.

Input size	Average runtime (secs)
128x128	128.20
256x256	1031.79
1024x1024	4211.58

1.2 Compiler optimisations

First, different compiler options were experimented with using two compilers: GCC 5.4.0 and Intel 2017.1.132. For each, three different sets of additional compiler flags were trialled: `-O3`; `-Ofast`; `-Ofast -mtune=native`. Each was run on the 128x128 input and the average runtime taken, listed in Table 2.

The fastest average times used the Intel compiler with the `-Ofast` and `-Ofast -mtune=native` flags, respectively. The negligible time difference between these sets of flags can be attributed to noise. Therefore, the `-Ofast -mtune=native` optimisation flags were used to compile the program from this stage onwards.

Table 2: Average runtimes for the 128x128 input with different compilers and compilation flags in seconds.

Compiler	Flags	Runtime (secs)
GCC	-O3	32.92
GCC	-Ofast	28.01
GCC	-Ofast -mtune=native	28.31
Intel	-O3	25.68
Intel	-Ofast	24.99
Intel	-Ofast -mtune=native	25.06

1.3 Loop fusions

The gprof profiler was used to investigate which functions the program spent the most time running during execution with its original structure. The most significant three results were: `collision` (average 68.13% of runtime); `propagate` (average 16.42% of runtime); `av_velocity` (average 15.48% of runtime). Each of these functions, along with `rebound`, contained a nested `for` loop that iterated over the entire inputted grid with every time step. There are 40,000 time steps during the execution of the program in the given test cases, meaning the extra iterations of the grid contributed a significant amount of runtime. The function `collision` especially performed many calculations per grid cell per time step, accounting for its large percentage in the runtime of the program. It is unnecessary to iterate over the entire grid separately for each function every time step.

The loops of `rebound` and `collision` were fused through the implementation of an if-else statement in the `for` loop of `collision`. This executes the logic of `rebound` if the specified cell contains an obstacle and the logic of `collision` if it does not. The logic

of `av_velocity` was also combined with this by adding the summation of `tot_u` and `tot_cells` to the loop of `collision` and modifying the return statement so the new `reboundCollisionAVVels` function, and subsequently `timestep`, calculates and returnes the av velocities each time step instead of `av_velocity`. Unlike `rebound`, which was removed, the original `av_velocity` was kept in the code as it needs to be called separately by `calc_reynolds` at the end of the program.
The loop in `propagate` was not fused with the larger loop because this step needs to be completed before the calculations in `reboundCollisionAVVels` begin, or the the accuracy the program output will be impacted.
This reduced the serial runtime of the program with the 128x128 input to an average of **20.55s**, the 256x256 input to an average of **188.73s** and the 1024x1024 input to an average of **791s**.

1.4 Improving arithmetic in the loop

In the main nested loop of the new function `reboundCollisionAVVels`, the calculations `2.f * c_sq * c_sq` and `2.f * c_sq` were each performed nine to eighteen times per loop iteration, meaning recalculating the same numbers nine to eighteen times per grid cell per time step, which was wasteful. Instead, these calculations were moved prior to the loop and stored as the constants `c_sq_sq_2f` and `c_sq_2f`. This means they are now each calculated only once per time step.
This had little impact on the 128x128 input, which remained at an average runtime of **20.57s**. However, it reduced the 256x256 input runtime to an average of **165.75s** and the 1024x1024 input to an average runtime of **714.53s**. This brought the non-vectorised serial implementation well within the ballpark times.

2 Vectorisation

2.1 Structure of arrays

To enable effective vectorisation by the compiler, the original array of structures of the Lattice-Boltzmann code was changed to a structure of arrays format. This was achieved by converting the struct `t_speed` from a single array of speeds per grid cell to an array of arrays. Each nested array stores one speed for each grid cell in the entire grid, from speeds 0 to 8. So the overall array in the new `t_speed` stores, in pseudocode: `[[array of speed 0s], [array of speed 1s], ... , [array of speed 8s]]`. This structure of arrays data layout is more conducive to vectorisation.

2.2 Assisting the compiler

A number of steps were taken to assist the compiler in vectorisation. The function used to allocate memory for the new arrays in `cells` and `tmp_cells` was changed from `malloc` to `_mm_malloc`, used to align the data along a boundary of 64 bytes—the length of a cache line. The lines `__assume_aligned(cells, 64)` and `__assume_aligned(tmp_cells, 64)` were added to the relevant functions to ensure the compiler recognised this. Additionally, `__assume((params.nx)%n==0)` and `__assume((params.nx)%n==0)` where n was replaced with powers of 2 up to 16 were included so the compiler could unroll the loops more efficiently.
Furthermore, `const` and `restrict` were included everywhere possible, the latter especially so the compiler would know `cells` and `tmp_cells` wouldn't alias in the loops. The directives `#pragma omp simd` and `#pragma omp simd aligned(cells, tmp_cells)` were included before all relevant inner `for` loops, so the compiler knew vectorisation was desired and would ignore remaining aliasing concerns.

2.3 Scalar assignment

Despite this, many loops in the code still did not vectorise. Therefore the compiler flag `-qopt-rpt` was used to investigate further.
The report stated the main loop had still not vectorised due to its use of scalar assignment, i.e. the summing of `tot_cells` and `tot_u` across the iterations of the loop. The directive before the loop was therefore changed to `#pragma omp simd aligned(cells, tmp_cells) private(tot_cells, tot_u)`, which finally enabled vectorisation.

2.4 Serial vectorisation results

However, the vectorised runtime on the 128x128 input was only reduced to an average of **11.7s**, which did not meet the ballpark time of **6.2s**.

Similarly, the vectorised serial implementation produced an average runtime of **283.36s** on the 256x256 input, compared to the ballpark of **48s**, and over **2400s** on the 1024x1024 input, compared to the ballpark **269s**. Therefore, without further improvement, it would not make a suitable foundation for the parallel implementation, so the decision was made to instead base the parallel implementation on the non-vectorised optimised serial implementation.

3 Parallel Implementation

3.1 Parallelising for loops

The program was then parallelised using OpenMP. This involved including `#pragma omp parallel for` before the majority of the `for` loops in the code, so the grid being iterated over would be divided amongst a specified number of threads.

3.2 Loop collapse

The majority of the loops in the program are nested loops. Simply adding `#pragma omp parallel for` would only parallelise the outer loop. Therefore, the `collapse(2)` directive was added to create the directive `#pragma omp parallel for collapse(2)` for each nested loop. At compilation time, this collapses the nested loop into one that can then be parallelised more effectively between multiple threads.

3.3 Reductions

However, this parallel implementation caused the time taken for the 128x128 input on 28 cores to increase to around **40s**. This was because of the summing of `tot_cells` and `tot_u` within the main loop of `reboundCollisionAVVels`. As these numbers had to be consistent across all threads, the threads would be forced to wait and synchronise every time these variables were incremented.

This problem was overcome with reductions. The directive at the beginning of the `for` loop in `reboundCollisionAVVels` and `av_velocity` became `#pragma omp parallel for collapse(2) reduction(+:tot_cells) reduction(+:tot_u)`, meaning `tot_u` and `tot_cells`' values across the different threads would be summed together at the end of the nested loop's execution. This brought the parallel runtime of the program on the 128x128 input over 28 cores to **2.3s**.

3.4 Non Uniform Memory Access

While OpenMP operates on the basis of all threads having a shared memory, the stored data operates on a 'first touch' principle where, when physically allocating memory, a thread allocates data to memory close to the core it is running on at the time. When threads are split across cores later to read and modify that data in memory, the thread acting on a specific part of that data is not necessarily running on the same core as the thread that allocated the data in the first place. Therefore, some data may be physically allocated further from the cores that actually operate on them than others, slowing data retrieval and therefore runtime.

To overcome this, threads can be pinned so that the threads that initially allocate memory for a certain data structure in parallel become the same threads that operate on that section of memory later on, subtly reducing runtime. This is achieved through setting the environment variables `OMP_PROC_BIND=true` and `OMP_PLACES=cores`. These commands bind processes to places and set those places to be the cores of the CPU.

Setting these variables correctly reduced the average parallel runtime on 28 cores to **2s** on the 128x128 input, **11.80s** on the 256x256 input, and **43.91s** on the 1024x1024 input.

4 Evaluation

4.1 Scalability

The graph in Figure 1 demonstrates the parallel efficiency of the parallelised program in comparison to the optimised serial version of the program for all test inputs. The parallel efficiencies were calculated as $PE = (s \div n) \times 100$ where n is the number of cores and s is the speedup, calculated as $s = oldtime \div newtime$. The new time for each number of cores per input size was taken as an average of three tests.

This figure demonstrates that while the efficiency gained through the parallel implementation quickly declines for smaller inputs, it scales well to 28 cores for the larger 256x256 and 1024x1024

grid inputs. However, it is only at most 60% efficient for the given test cases as parts of the program still rely on serial execution. It is also less efficient than the optimised serial implementation on 1 core due to the overhead induced by loop collapses and reductions the serial implementation does not make use of.

4.2 Roofline analysis

Figure 2 shows the roofline graph for the parallel implementation on 28 cores on the 128x128 grid, generated by the Intel Advisor tool. It shows that most of the functions of the program still do not maximally utilise memory bandwidth. However, significantly the loop inside `reboundCollisionAVVels`, where the program spends a majority of its runtime—represented by the red marker—is past the ridge point of the L1 cache roofline. This means the most significant part of the program has become compute bound rather than memory bandwidth bound for data small enough to fit in L1 cache. It has also reached the scalar roofline for L1 cache, meaning it reaches theoretical maximum performance on these smaller input sizes when there is no vectorisation.

Additionally, while it is still memory bandwidth bound by the DRAM roofline, the loop in `reboundCollisionAVVels` has achieved the current maximum possible amount of GFLOPs for grid sizes too large to fit into L3 cache. Therefore, for very large and very small grid sizes, the most significant part of the program has has achieved current possible maximum GFLOPS allowed by respective limiting resources.

5 Conclusion

The optimised serial implementation successfully surpassed the given ballpark runtimes, though the vectorised serial did not. The parallel implementation, while also not as successful, demonstrably scales effectively for large inputs and sped up the runtime of the original Lattice-Boltzmann code on the 1024x1024 grid an average of **95.53x**. While there is room for improvement, it has also reached maximal scalar efficiency for inputs small enough to fit in L1 cache. Therefore, the Lattice-Boltzmann code has been successfully optimised with OpenMP.

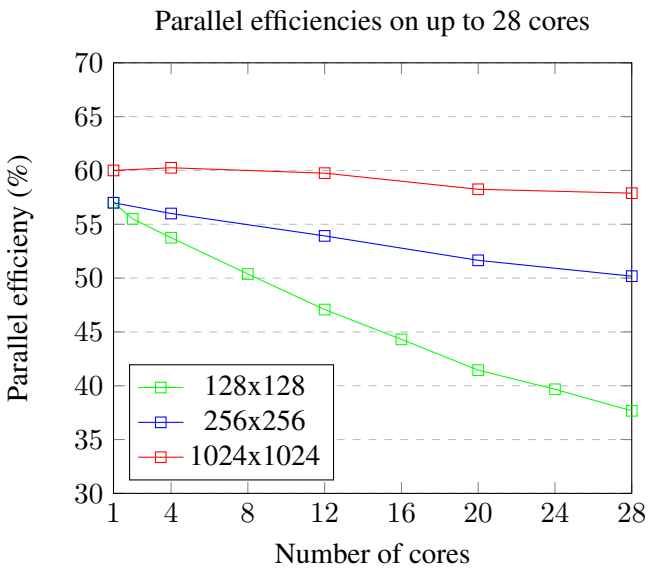

Figure 1: Parallel efficiency of the parallel implementation on each input on 1 to 28 cores, compared to the optimised serial implementation.

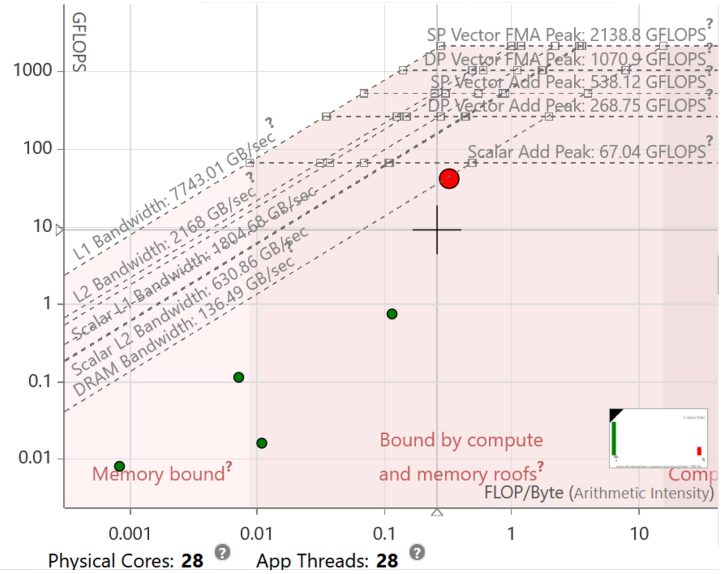

Figure 2: Roofline analysis of the 28-core parallel implementation on Blue Crystal Phase 4.