AHPC - Lattice Boltzmann OpenMP

Jerry Wang (sw16437)

I. Compiler Flags

This is a report on the optimisation of the serial and OpenMP implementation of lattice boltzmann algorithm. The timing for the original code before any optimisation are listed below in Table I.

size	128*128	256*256	1024*1024
time	40.2	310.3	1298.1

TABLE I
ORIGINAL SERIAL CODE RUNTIME

Before making any change to the code, it is worthwhile to make sure the program is compiled with the most adequate optimisation flags. By using ICC instead of GCC and adding optimisation flag `-O3 -fast`, the program saw an immediate speed improvement of 1.2x.

II. Arithmetic

To avoid repeated calculations, indexes used to access the grids such as $ii + jj * params.nx$ are replaced by a variable that is only calculated once per iteration. Since float divisions are costly, $w0$, $w1$, $w2$ are also replaced with constants. The calculation of equilibrium densities can be simplified by changing divisions of fraction into multiplications. This provided a speed-up of about 1.1x over the previous step.

size	128*128	256*256	1024*1024
time	28.6	236.1	1017.9

TABLE II
RUNTIME AFTER OPTIMISING ARITHMETIC

III. Structure of Arrays

The original code stored the cells in array of structures. Each cell in the grid is an individual structure storing 9 speeds, which is then stored in an array. To prepare for vectorisation, the code needs to be transformed to storing cells in structure of arrays. To achieve this, `t_speed` is modified to contain `float* [NSPEEDs]`. Each of the pointers points to an array containing one direction of speeds from all cells. This will notably suit vectorising the inner loop because now the speeeds of a single element can be accessed in 9 arrays by the same index. After applying the above step, we observe an immediate slow down of runtime. This is expected as the original array of structure has better data locality.

size	128*128	256*256	1024*1024
time	34.2	289.3	1218.0

TABLE III
RUNTIME AFTER APPLYING SOA TRANSFORMATION

IV. Vectorisation

In order to take advantage of the new SOA structure, adequate vectorisation techniques are required. This includes using `_mm_malloc()` and `mm_free()` for memory management, using `_assume_aligned()` to hint compiler of particular memory alignment. Intel vTune Amplifier Advanced Hotspot also shows that the `for` loops used for calculating local densities and relaxation steps are a major source of slowdown. This is possibly because these nested for loops are preventing auto-vectorisation. To fix this issue, local density calculation is expanded to a single statement with 9 addition operations, and relaxation steps are calculated individually in 9 standalone statements instead of in the for loop. This provided around 3.5 seconds of improvement. `#pragma ivdep` is used on the inner loops to inform of compiler of independence between iterations. Finally, `#pragma omp simd` are used on the inner loops to enforce vectorisation. This step produced a major boost in speed of 1.4x. (Table IV

size	128*128	256*256	1024*1024
time	24.1	206.4	852.7

TABLE IV
RUNTIME AFTER APPLYING VECTORISATION

V. Loop Fusion and Pointer Swap

During each time step, the grid is iterated through a few times. This is costly in terms of memory bandwidth. Therefore, reducing the reading and writing of the grid should provide considerable performance improvement. As a result, `propagate()`, `rebound()` and `collision()` are combined into a single set of double for loop, writing only once from the original grid to the temporary grid per time step. And the pointer of the grids are swapped after each iteration to make sure the updated grid is always used as the source grid in the next iteration. However, this did not produce the expected improvement. Using Intel vTune Amplifier, it can be seen that the most time inside the newly merged `collision()` function is introduced by the `if` statements from `propagate()` and `rebound()` that are used to check if an element contains any obstacle. This

makes sense since having `if` statements can potentially break the auto vectorisation by the compiler. Considering Lattice Boltzmann method is a memory-bounded task, it is reasonable to give up some computation power for reduced memory bandwidth usage. Thus, we can remove the `if` statements, meaning calculating each element both as an obstacle and not an obstacle, and only evaluate if it is an obstacle upon the final assignment to the destination grid using an conditional operator. By checking the vectorisation report, we can confirm that this solution is vectorised and correctly combines `propagate()`, `rebound()` and `collision()`. This produced a tremendous improvement of 2x.

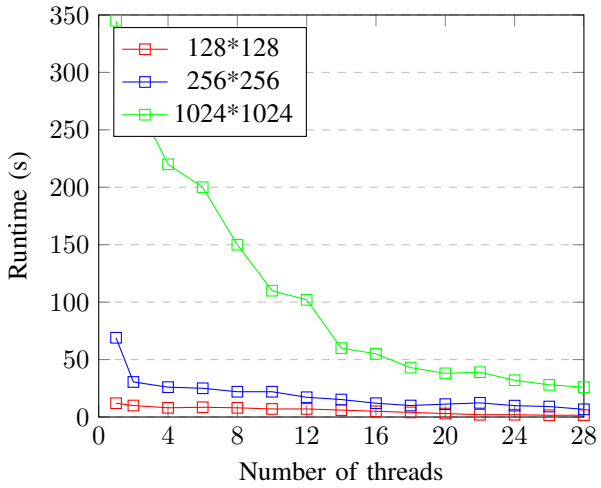
size	128*128	256*256	1024*1024
time	12.2	49.1	176.8

TABLE V
RUNTIME AFTER APPLYING LOOP FUSION

VI. OPENMP

To implement OpenMP, `#pragma omp parallel for` is added to the newly combined fusion loop, `av_velocity()` and `initialise()`. This makes sure each OpenMP thread initialises a chunk of grid closer to their proximities. This helps the program produce a runtime close to the ballpark timings. It is worth noting that making smaller functions such as `accelerate_flow()` increases the runtime very noticeably. This could be that the benefits of parallelisation is overwhelmed by forking and re-joining overheads. Another useful OpenMP clause is `reduce`. By applying reduce in `av_velocity()`, we can observe another decent decrease in runtime by 1.2x. Using `OMP_PROC_BIND=close` bring slight further improve. The final timings after applying all the techniques in this section are shown below. (Table VI)

OpenMP runtime on all 28 cores



It can be observed in the graph that while increasing the number of OpenMP threads does initially decrease runtime, the scaling is better on larger grids, while the performance of smaller girds plateaus very quickly. This is expected since

size	128*128	256*256	1024*1024
time	1.5	6.7	27.9

TABLE VI
RUNTIME USING 28 OPENMP THREADS

using 28 threads on smaller grids such as $128 * 128$ may result in a competition in memory bandwidth and under-utilising each thread. We can also observe a speed-up around 14 threads particularly for the largest grid. This could be because the largest grid only fits in L3. Since using more than 14 threads requires the support of 2 sockets, that effectively doubles the available L3 cache. When the grid can fit in the larger L3 cache instead of DRAM, we can expect a decent improvement in runtime. The smaller grids do not experience the same boost in speed because both the $128 * 128$ grid and the $256 * 256$ grid can fit in the L2 cache provided by a single socket.

We can also observe some fluctuations and drop in performance around 6 cores, 16 cores and 26 cores. This could be because of load imbalance. When the total iteration count of the loops can not be evenly divided by the number of threads, one thread will be doing less work than the others. To address this, we can use the `nowait` clause in the pragma. Although this seems to provide some speed-up, they do not scale as good as other thread counts that can evenly divide the total loop iteration numbers.

VII. ROOFLINE ANALYSIS

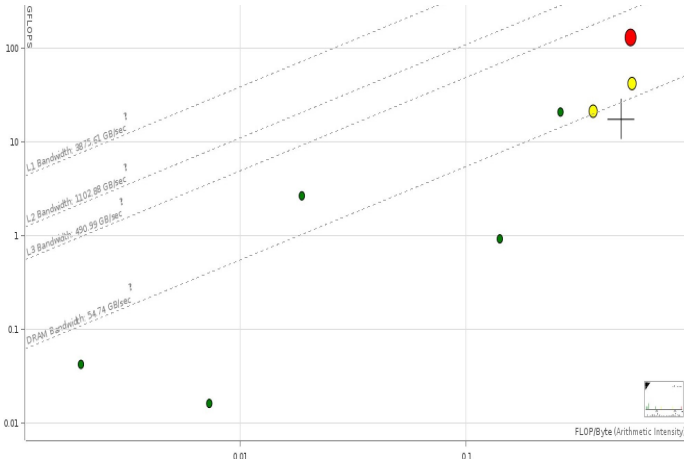


Fig. 1. Roofline model of BC4, 28 cores

From the roofline graph, we can see that the OpenMP achieved a performance of 20.1 GFLOPS, much higher compared to the original serial code.

Overall, we see about 42x of speed-up compared to the code before optimisation. And optimisations that aims to reduce memory bandwidth usage provide more performance gains. Despite the variance of performance from BlueCrystal, the final timings adequately match the ballpark timings.