Serial Optimisations and OpenMP Parallelism on 2D Lattice Boltzmann Code

Luke Zhang (az16408)
Department of Computer Science
University of Bristol
Bristol, United Kingdom

Abstract

***Abstract*—This experiment explores and optimises a 2D Lattice Boltzmann (LB) code via a variety of optimisation techniques. The program is run on a supercomputer to utilise the multi-core processor architecture on a computing node. With serial optimisation and vectorisation, all three inputs gained a considerable speed-up in performance. Furthermore, an enormous 50-55x speed-up for both the 256x256 and 1024x1024 inputs is measured by parallelising our serially optimised and vectorised code. It is also found that the existence of NUMA effect produces different scaling graphs.**

***Index Terms*—Serial Optimisation, Vectorisation, SIMD, OpenMP, NUMA**

I. Introduction

With Moore's Law coming to an end and the rise of multi-core computing architecture, there is an increasing need to optimise programs that are extremely computationally intensive with very large inputs. High Performance Computing (HPC) is one of the main research fields when it comes to applications such as particle simulations, space research [1] and novel drug discovery. [2]. In this case, there are three main optimisation techniques applied to build faster fluid simulation for the LB code, namely serial optimisation, vectorisation and OpenMP parallelization.

II. Experimental design

A. Blue Crystal Phase 4 Processor

In this experiment, we use Blue Crystal Phase 4 (BC4) supercomputer managed by University of Bristol HPC group. The supercomputer's main processing unit is comprised of Lenovo nx360 m5 compute nodes, where each node has two Xeon Intel E5-2680 v4 Broadwell CPUs. In terms of cache hierarchy, there are L1 data and instruction caches per core, and 14 cores in each CPU. There are also L2 caches per core, and a single but much larger L3 cache per CPU which is shared by all the cores.

B. Serial Optimisation

The first part of this experiment is serial optimisation, which uses a single core on BC4. For this part, we use the LB code [3] provided by HPC group at University of Bristol to compute a baseline. Then, various optimisation techniques can be used to improve the efficiency of serial executions. From there, the serially improved version can be bench-marked against the baseline. It is noteworthy that the data layout is left as the original Array of Structure (AoS). This is because this type of layout has better data locality due to the 64 bytes cache lines on Blue Crystal 4, hence suites the scalar and non-vectorised version of the code well.

1) ***Compiler and Optimisation Flags****:* One way of helping the compiler optimise a program is to pick a good compiler and suitable optimisation flags. Here the 2020 version of Intel's ICC compiler is used. This is because this version of ICC compiler produces much faster run-time than the default GNU's GCC 5.4.0 compiler. As for optimisation flags, `-Ofast` is added to enable aggressive optimisation to gain more speed. This flag contains `-O3` level optimisation and many other flags. Furthermore, the optimisation flags also include `-xHost` which can optimise for the platform we are compiling our code on. Last but not least, advanced vector extension `-xVAX2` will also be included in order to vectorise the code efficiently in the following section.

2) ***Loop Fusion****:* In this step, we fuse the three functions that reside in the main `timestep()` function, namely `propagate()`, `rebound()` and `collision()`. Particularly, this is done to avoid iterating through the same loop within these functions separately and reduce loop overhead. It worth noting that the order of executions for these three functions needs to be correct due to data dependencies. Additionally, `av_velocity()` can also be merged into the same loop. This way the whole grid can be iterated over once per timestep.

*3) **Pointer Swap**:* To further optimise the program, the number of read and write can be reduced via pointer swap. The original program reads from `cells` and writes to `tmp_cells` in `propagate()`, then reads from `tmp_cells` and writes to `cells` in `rebound()` or `collision()`. This can be simplified by reading from `cells` once in `propagate()`, followed by some calculations and then write to `tmp_cells` in `rebound()` or `collision()` depends on if there is an obstacle or not for each iteration. At the end of each timestep, a pointer swap between `tmp_cells` and `cells` are performed to obtain the correct values.

*4) **Arithmetic Improvement**:* One more thing that can be improved is arithmetic. Generally, division costs more clock cycles than multiplication, this cost can accumulate in loops with large number of iterations. Therefore, it's best to use as many multiplications as possible to reduce the time spent on arithmetic. In this case, divisions are manually replaced by multiplications in the main timestep.

C. Vectorisation

In this part, we stick to a single core and add vectorisation to our program.

*1) **Compiler Flags and Keywords**:* As the Intel compiler comes with advanced vector extension, the flag `-xVAX2` is enabled. To preventing `cells` and `tmp_cells` aliasing with each other, the `restrict` keyword is enforced. What's more, the `const` keyword is used everywhere so that the compiler knows these variables stay the same throughout each iteration.

*2) **Data Alignment**:* To assist the compiler in vectorising the program, there are various techniques can be utilised to tell the compiler that our data is aligned. Here, we use `_mm_malloc()` when allocating memory for our data, then `_assume_aligned()` right before we use certain pointers (mainly `cells` and `tmp_cells`). For the alignment we use 64 bytes as this matches the cache line on BC4. In addition, we can also use `_assume()` to give more information about loop counters. This can help the compiler unroll loops more efficiently during vectorisation.

*3) **SIMD**:* Vectorisation is essentially data parallelization. The BC4 CPU supports 256-bit wide AVX Simple Instruction Multiple Data (SIMD) instructions, which can be used to vectorise our code. For vectorising our algorithm more efficiently, we use a Structure of Array (Structure of Array) data layout for this part. The reason is that this data layout is more efficient when using SIMD. As multiple iterations of a loop can be executed concurrently, this in turn can maximise the use of our data and improve the computation efficiency. In this case, the SIMD pragma (`#pragma omp simd`) is mainly used in the inner loops inside the time step function as this is where the program spends most of its time.

D. Parallelization with OpenMP

Finally, we take our serially optimised and vectorised code and parallelise as many tasks and data as possible. This is done by scaling our program from 1 core up to 28 cores. From there we can also obtain a scaling comparison by using different scaling approaches.

*1) **Pragmas and Reductions**:* To parallelise our program, we use the pragma `#pragma omp parallel for` right above the loops that we want to parallelise. In this scenario, it is expected that this pragma will gain most speed-up in the main `timestep()` function. For updating a shared variable in parallel, we use reductions along with the parallel for pragma. The reason for using reductions is because each thread will compete to update the shared variable. This will very likely results in incorrect update of the shared variable. Therefore, reductions can be of use to ensure only one thread update a shared variable at a time. In our case, the `tot_u` is a shared variable in the main loop. This means a reduction clause `reduction(+:tot_u)` can be placed right after the parallel for pragma to update it correctly.

*2) **NUMA-awareness**:* Generally speaking, caches between cores in L1 and L2 are not shared as they are private to each core. They only interact for coherency to make sure the copies of the data stay up to date when cache lines are updated. However, L3 cache is shared with all cores on the socket. This may cause the same data being touched at different times. As a result, we would expect some Non-Uniform Memory Access (NUMA) effects when accessing data in the other socket's L3 cache. To make sure our code is NUMA aware, we parallelise the initialisation routine as well as the main `timestep()`. This should decrease the number of remote memory access. It is also expected that the run-time would vary less from many different runs.

However, we cannot rely solely on the above setting to produce reliable NUMA aware program. This is due to the possibility that the OS might still be able to move threads from one core to another. To prevent this from happening, we need to set thread pinning policies. This can be achieved by using the environment variables `OMP_PROC_BIND=true` and `OMP_PLACES=cores` in our environment. The former ensures that OS will not move our threads and the latter guarantee each place is a single core that contains one or more threads.

*3) **Thread Pinning Policy: Close vs Spread:*** Following the observation of NUMA-awareness, we can take one step further to see how the program will behave when using different thread pinning policies. This can be accomplished by setting `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread` respectively. As we scale up with more cores, the former filling up cores one socket (14 cores) at a time whereas the latter filling up cores back and forth between the two sockets. It is anticipated that we would get a clear jump in performance after 14 cores with `OMP_PROC_BIND=close` due to non-NUMA effect before 14 cores.

III. Results

There are more data live in DRAM as the size of the data grows to 1024x1024, while the smaller inputs will live in the on-chip caches. Here we use 256x256 as a fixed input and compare the difference in roofline between serially optimised and vectorised version of our code using ICC. It can be shown in Fig 1 that our program is mostly memory bounded by L3 bandwidth when only serial optimisation is applied. Vectorisation enables us to move up and become more L2 bandwidth bounded. This is likely because of the increase of the arithmetic intensity which leads to a much better performance.

A. Roofline Analysis

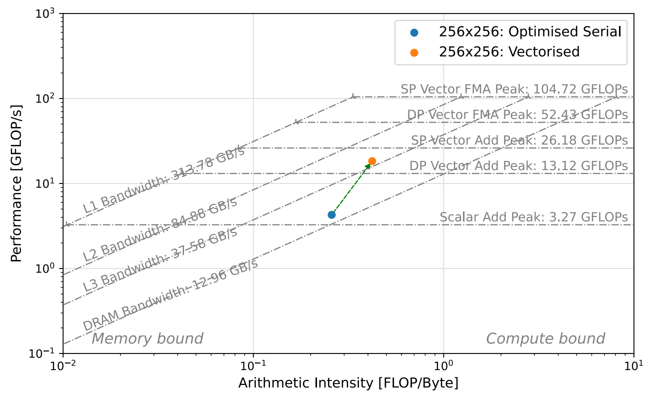

Fig. 1. Roofline Analysis: Optimised Serial vs Optimised Serial+Vectorised for 256x256 input on one core with ICC.

TABLE I
Run-Time Comparison with ICC: Optimised Serial(1 core) vs Optimised+Vectorised Serial(1 core) vs OpenMP(28 cores)

	Final Elapsed Total Time (s)		
	Optimised Serial	Vectorised Serial	OpenMP
128x128	20.02	6.16	0.85
256x256	159.62	42.45	2.82
1024x1024	671.70	218.68	13.45

B. Performance Comparison

The speed-up resulted from switching the compiler is very noticeable. Here the run-time baselines for the original code with GCC are 29.01s, 231.73s and 974.27s for 128x128, 256x256 and 1024x1024 respectively. The performance with the three optimisation methods using ICC can be seen in TABLE I. After optimising serially, the version compiled with Intel is 1.45x faster than GNU for all three inputs. This is because that there are more optimisations behind the scenes in the Intel compiler. Additionally, applying loop fusion, pointer swap and arithmetic improvement in section II-B make loops more efficient, enable much faster data manipulation and calculation.

Once vectorisation has been implemented, there are even more performance gain can be observed. From the compiler optimisation report, it is shown that the Intel compiler is able to vectorise many loops in places where GNU is not. This results in a drastic speed-up of 4.7x for 128x128, 5.46x for 256x256 and 4.46x for 1024x1024. The SIMD exercise the same instruction on multiple data at the same time. This results in the processor doing more within the same number of clock cycles which gives more speed-up. It is worth to note that this time the speed-up in performance does not scale linearly as input grows. This is because the program become more memory bounded as we increase the size of the input.

Moreover, it can be seen that parallelization with 28 cores elevates the gains even further. With ICC and OpenMP achieving significant speed-up of 34.13x, 82.17x and 72.43x compared to the baseline with GNU for all three inputs ranging from smallest to largest. Thanks to the power of multicore CPUs, both tasks and data can be parallelised which accelerate the execution time tremendously.

C. Scalability of OpenMP Parallelism

In this section, we compare the scalability of our parallelised code against the optimised serial version with the Intel compiler. From the scaling graph in Fig

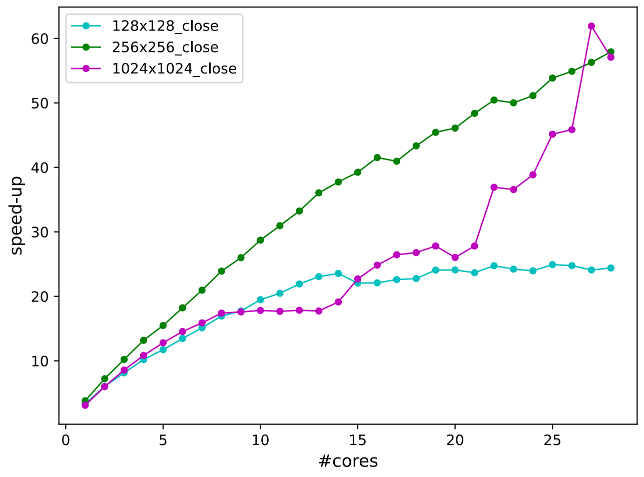

Fig. 2. Scaling with OMP_PROC_BIND=close: 1-28 cores

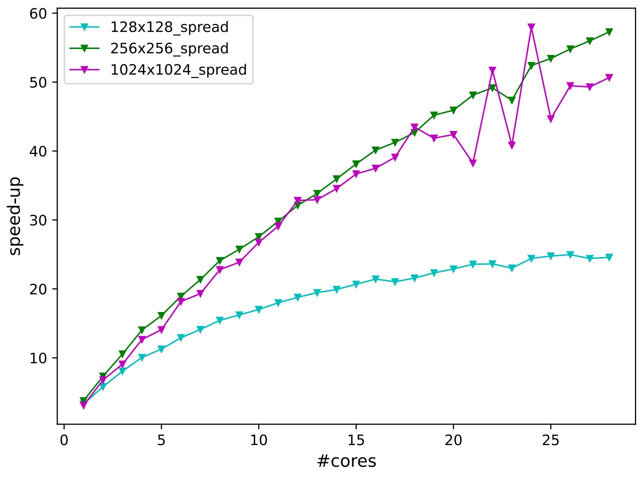

Fig. 3. Scaling with OMP_PROC_BIND=spread: 1-28 cores

2, it can be shown that the run-time with input size of 256x256 and 1024x1024 scale up very nicely. Both of them gained a dramatic speed-up of above 55x when running with 28 cores.

Additionally, we can see a plateau of speed-up for 1024x1024 before 14 cores, followed by a distinctive jump in performance from 14 to 15 cores. This is because the `OMP_PROC_BIND=close` pinning method does not have NUMA-awareness before the number of cores switch from 14 to 15. As a result, switching from using one socket to two give extra memory bandwidth for large input. However, the 128x128 is an exception in this case despite of the steady improvement and eventual 25x speed-up. The reason for this is likely because the small input size does not benefit from the additional memory bandwidth, which means it is more compute bounded. What's more, this also results a diminishing return in performance as more cores being added for small input size.

As for the scaling graph with `OMP_PROC_BIND=spread` in Fig 3, it can be observed that a similar trend unfolds for the 256x256 and 128x128 inputs compared to the one in Fig 2. However, the transition for the 1024x1024 appear to be not the same. There is more speed-up with more cores added this time regardless of the lower eventual 50x performance. As both sockets are being used when scaling up our code, our program is NUMA aware. Consequently, there is no sharp jump in performance when switching from 14 cores to 15 cores for all three inputs. This also means that the transition is smoother since there is no extra memory bandwidth benefit.

IV. Conclusion

In conclusion, this experiment explored and conducted various methods of optimising a given LB code. For the serial optimisation result, the program compiled with ICC gained 1.45x in performance for all three inputs compared to the baseline with GCC. After applying various techniques to help with vectorisation, we achieved a considerable speed-up of 4.7x for 128x128,5.46x for 256x256 and 4.46x for 1024x1024. In the OpenMP parallelization section, with all 28 cores on a node we obtained a 50-55x speed-up for both 256x256 and 1024x1024 inputs and 25x speed-up for 128x128 input. Finally, we see that the `OMP_PROC_BIND=spread` has a smoother scaling than the `OMP_PROC_BIND=close` for large input, with no sudden jump in performance transitioning from 14 to 15 cores. This informs us that the thread pinning and allocation is working as expected due to the presence of NUMA effect.

References

[1] "14 high performance computing applications & examples," 11 2019. [Online]. Available: https://builtin.com/hardware/high-performance-computing-applications

[2] T. Liu, D. Lu, H. Zhang, M. Zheng, H. Yang, Y. Xu, C. Luo, W. Zhu, K. Yu, and H. Jiang, "Applying high-performance computing in drug discovery and molecular simulation," *National Science Review*, vol. 3, no. 1, pp. 49–63, 01 2016. [Online]. Available: https://doi.org/10.1093/nsr/nww003

[3] S. M. Smith, T. Deakin, and H. Waugh, "Coms30006 - advanced high performance computing - lattice boltzmann," 2022. [Online]. Available: https://github.com/UoB-HPC/advanced-hpc-lbm