Serial optimisations & OpenMP parallelism

Name: Xinran Wang
student number: 1924931

1. Introduction

This report details the optimization of the Lattice Boltzmann Method (LBM) code. Initially, serial optimization techniques, including loop fusion and pointer swapping, were applied. To facilitate code vectorization by the compiler, data structures were modified, and memory alignment techniques were employed. The final stage of optimization involved using OpenMP for parallel acceleration, scaling from a single core to 28 cores to enhance performance and efficiency.

2. Serial Optimization

2.1 Implementation

Upon reviewing the initial code, it was observed that in each cycle, five distinct functions were executed: **accelerate_flow()**, **propagate()**, **rebound()**, **collision()**, and **av_velocity()**. Notably, among these, **propagate()**, **rebound()**, **collision()**, and **av_velocity()** were each responsible for iterating through the entire grid once. This process involved multiple loads of **cells** and **tmp_cells** from memory. Given that accessing data from main memory requires 200-400 cycles, whereas cache (specifically L3 cache) access only needs 20-30 cycles, the approach was to merge these four functions to necessitate only a single iteration over the entire grid. During the merge of the **propagate()** function, a nine-element array was used to temporarily store necessary data, reducing unnecessary memory consumption and avoiding data conflicts. This merger slightly improved the arithmetic time complexity from $3O(n^2)$ to $O(n^2)$.

Furthermore, pointers were used to avoid copying data through double for loops. Pointer swapping, by merely changing data references rather than moving data itself, significantly reduced the need for data movement; it also maintained a high cache utilization rate by avoiding frequent cache line replacements.

To optimize the computational process, it was noted that calculations within **av_velocity()** accounted for 19.4% of the total runtime. To optimize this, constants and repeatedly calculated values were computed in advance and passed to the function as variables, aiding compiler optimization. Further analysis revealed that the primary computational load in **av_velocity()** stemmed from the **sqrtf()** function, which occupied 18.1% of the total runtime. An initial attempt was made to replace this function with a faster, albeit less accurate, method using Newton's iteration. However, this approach was abandoned due to poor performance, because modern CPUs are highly optimized for floating-point operations, including **sqrtf()**, with specialized instructions (such as **AVX**) allowing rapid execution.

Finally, debugging revealed that most cells do not require the **rebound()** operation, indicating most cells do not contain an obstacle. To improve branch prediction performance, the order of if statements was modified to execute the most frequent operation immediately following the if statement, with less frequent operations in the else branch. This improves performance since incorrect branch predictions cause the CPU to backtrack and re-execute the correct branch, leading to efficiency reductions.

Table 1: Compiler Flag Performance

Flag \ Grid Gize	128x128	128x256	256x256	1024x1024
-O3	30.35s	61.32s	245.96s	930.25s
-Ofast	24.04s	51.74s	187.18s	783.65s
-Ofast -xHost	23.86s	47.42s	185.19s	771.28s
-Ofast -xHost -qopenmp	22.12s	42.15s	180.23s	733.78s

Note: Compiler is ICCversion2017.1.132. Each data point is the average of five runs.

Table 2: Compiler Version Performance

Compiler Version \ Grid Size	128x128	128x256	256x256	1024x1024
GCC2016	23.88s	47.92s	184.15s	768.23s
GCC2017	23.25s	45.28s	182.53s	750.40s
ICC2020	23.75s	46.87s	183.98s	756.79s
ICC2017	22.12s	42.15s	180.23s	733.78s

Note: Flag are all -Ofast -Xhost -qopenmp. Each data point is the average of five runs.

2.2 Performance Analysis

Following the optimizations and after comparing compiler versions and flags as illustrated in **Table 1** and **Table 2**, the best runtime results were obtained using the Intel C++ Compiler (**ICC**) version 2017.1.132, with the compilation command:

icc-std=c99-Ofast-xHost-qopenmp

The runtimes for grids of different sizes were as follows: 22.12 seconds for a 128x128 grid, 42.15 seconds for a 128x256 grid, 180.23 seconds for a 256x256 grid, and 733.78 seconds for a 1024x1024 grid. The compiler flags are explained below:

-xHost: This flag enables optimizations for the current CPU architecture. It automatically detects the processor type and enables the most suitable optimization settings. This means that the binary file generated will perform optimally when run on the current machine.

-Ofast: This enables all optimizations of **-O3**, along with further approximations for mathematical operations and optimizations for floating-point numbers.

-qopenmp: This enables the compiler's automatic vectorization capabilities and parallel processing features.

The compiler analysis report, generated due to the **-qopt-report=5** flag, revealed that the reason the Intel 2017 compiler achieved faster execution times was its ability to automatically vectorize specific computations within the **fusion_more()** function, notably the loop calculating **local_density**, even though the code was not explicitly written for vectorization.

3. Vectorization

3.1 Implementation

To further enhance the code, I explored the use of vectorization, aiming to reduce the overall workload by using the processor's SIMD (Single Instruction, Multiple Data) instructions, allowing multiple identical operations to be executed simultaneously. This enables a single CPU to perform more tasks within the same timeframe, thereby reducing the consumption of computing resources overall.

For the compiler to vectorize the code, it was necessary to modify the data structure. The original structure was an Array of Structures (AoS), where each cell's speeds were stored consecutively, leading to non-contiguous memory access when, for example, SIMD operations were required on speed1 for all points. This lowered cache utilization and increased memory access latency. By transitioning to a Structure of Arrays (SoA), where each speed of all cells is stored contiguously, SIMD operations can seamlessly load data from memory, enhancing cache efficiency and data locality.

However, merely changing the data structure was insufficient for optimal compiler vectorization. I also applied the **restrict** qualifier to pointers and **const** to constants as extensively as possible. The restrict keyword explicitly declares that data access through pointers is independent and will not cause data conflicts, allowing the compiler to vectorize more confidently. Furthermore, during grid initialization, I utilized **_mm_malloc()** to ensure data alignment. For static arrays, alignment was achieved using declarations like:

float keep[9] __attribute__((aligned(64)));.

Proper data alignment ensures that a cache line can fully load the required data in one or fewer memory access cycles, reducing the number of accesses to the main memory and thus improving cache efficiency. With cache lines of 64 bytes on Blue Crystal, data was initialized with 64-byte alignment. Before using these data, **__assume_aligned()** was employed to indicate to the compiler that pointers are always aligned, allowing for more efficient memory access and instruction scheduling strategies, particularly when the compiler cannot infer data alignment.

Finally, I made extensive use of **__assume()**, such as using **__assume(params.ny % 2 == 0)**; before loops. Knowing the exact divisibility of **params.ny** enables the compiler to fully or partially unroll loops to reduce the number of iterations and enhance the execution efficiency of the code within the loop. This also eliminates the compiler's need for boundary condition checks, making it easier to apply vectorization instructions. To further ensure compiler vectorization of specific loops, I applied **#pragma omp simd** instruction, reinforcing the compiler's ability to vectorize those sections of the code.

3.2 Performance Analysis

Following the aforementioned optimizations and using Intel C++ Compiler (ICC) version 2017.1.132, with the compilation command:

icc-std=c99-Ofast-xHost-qopenmp

, the obtained runtimes for grids of different sizes were as follows: 7.12 seconds for a 128x128 grid, 15.56 seconds for a 128x256 grid, 57.78 seconds for a 256x256 grid, and 278.14 seconds for a 1024x1024 grid. The compiler's analysis report indicated that all loops within the fusion function were successfully vectorized. Moreover, a comparison with serial optimization, as shown in **Table 3**, reveals that vectorization further optimized execution speeds significantly. Particularly for a 1024x1024 grid, the vectorized runtime was substantially faster than the ballpark estimates.

Table 3: Performance Comparison: Serial vs Vectorization

Method \ Grid Size	128x128	128x256	256x256	1024x1024
Serial	22.12s	42.15s	180.23s	733.78s
Vectorization	7.12s	15.56s	57.78s	278.14s

Note: Each data point is the average of five runs.

4. OpenMP parallelism

4.1 Implementation

Previously, my code was running on a single core. To further enhance speed, it was necessary to leverage all cores on a node, utilizing two CPUs for a total of 28 cores. This was achieved by setting **OMP_NUM_THREADS=28**, indicating to OpenMP to create 28 threads for executing parallelizable code segments. The next step involved specifying the code segments for parallel execution.

I applied the instruction
#pragma omp parallel for collapse(2) reduction(+:tot_u, tot_cells) schedule(static)
to the loops iterating over the entire grid. The **#pragma omp parallel for collapse(2)** instruction implies that the subsequent two nested loops are merged into a larger iteration space, which is then divided and assigned to different threads for execution. Given that threads in this region need to share **tot_u** and **tot_cells** variables, and after analyzing their roles, it was found that using **reduction(+:tot_u, tot_cells)** allows for independent accumulation of **tot_u** and **tot_cells** within each thread. The accumulated results from all threads are then combined into these two variables at the end of the parallel region. This approach not only resolves data races and synchronization issues but also enhances efficiency compared to using **critical** or **Firstprivate**. The **schedule** clause specifies the strategy for distributing loop iterations to threads. **schedule** (**static)** scheduling means iterations are divided into equal chunks, by default the number of iterations divided by the number of threads (28).

I parallelized the initialization routines in the same manner to ensure that each thread allocates data during the initialization phase that it will also process during the computation phase. This enhances data locality, meaning each processor or core primarily accesses data in its local memory during computations, reducing the need for accessing remote memory, which has a higher latency. However, the operating system still has the authority to move threads from one core to another or even between different processor sockets during runtime. To prevent this, I used
OMP_PROC_BIND=true and
OMP_PLACES=cores.
OMP_PROC_BIND=true ensures that threads will not be moved. **OMP_PLACES=cores** instructs the OpenMP runtime system to place threads on separate processor cores. Implementing this ensures each thread runs on its own core, maximizing CPU resource utilization and helping to reduce resource contention between different threads.

4.2 Performance Analysis

Following all of the optimizations and using Intel C++ Compiler (ICC) version 2017.1.132, with the compilation command:
icc-std=c99-Ofast-xHost-qopenmp, the obtained runtimes for grids of different sizes were as follows: 1.18 seconds for a 128x128 grid, 2.92 seconds for a 128x256 grid, 10.43 seconds for a 256x256 grid, and 36.86 seconds for a 1024x1024 grid.

Table 4: Performance Comparison: Vectorization vs OpenMP

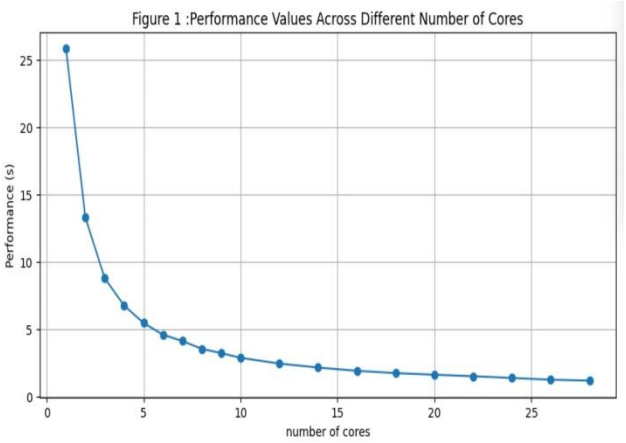
Method \ Grid Size	128x128	128x256	256x256	1024x1024
Vectorization	7.12s	15.56s	57.78s	278.14s
OpenMP	1.18s	2.92s	10.43s	36.86s

Note: Each data point is the average of five runs.

As indicated by **Table 4**, employing all cores for parallel execution significantly enhanced performance compared to using vectorization, though it remained slightly slower than ballpark estimates. To identify bottlenecks, I utilized compiler analysis reports and runtime analysis tools, attempting further optimizations through the following approaches:

1. Increasing the number of threads beyond the number of cores, for instance to 32, did not enhance performance but rather degraded it. This is because having more threads than cores will lead to thread switching on some cores, as a core can execute instructions from only one thread at any given moment. Thread switching involves saving the current thread's state (such as registers and program counter) and loading the next thread's state, incurring overheads that outweighed the computational benefits, thus reducing performance.
2. I experimented with different scheduling methods, including dynamic, auto, runtime, and guided. However, results indicated that using static scheduling to evenly distribute tasks among threads is the optimal choice.
3. I assessed whether unnecessary thread synchronization points existed within my code, as both implicit and explicit thread synchronizations introduce additional overhead. It was determined that my code did not contain unnecessary synchronization points, ensuring that the overhead associated with synchronization was minimized.
4. Reducing the number of threads was also considered. In **Figure 1**, a performance analysis across a range from 1 to 28 threads showed increasing speed, confirming that aligning the number of threads with the number of cores remains the optimal strategy. This approach maximizes CPU utilization without incurring the costs associated with thread management and context switching, thus ensuring efficient parallel execution.
5. The results were obtained through performance analysis tools, which revealed that, despite some functions not fully exploiting memory bandwidth, the principal function, **fusion_more()**, has exceeded the L1 cache's roofline. This indicates that for datasets small enough to fit entirely within the L1 cache, the function has become compute-bound rather than being limited by memory bandwidth. Moreover, it has achieved the scalar peak of the L1 cache, implying that, in the absence of vectorization,

it has reached the maximum theoretical performance for these smaller input sizes. For datasets too large to be fully contained within the L3 cache and thus still constrained by DRAM bandwidth, the loops within **fusion_more()** have managed to achieve the highest GFLOP/s performance feasible under the existing conditions. This illustrates that for both significantly large and small data sizes, this key part of the program has realized the maximum GFLOP/s performance allowed by their respective constraining resources.



5. Conclusion

In this report, I extensively explore the optimization process of Lattice Boltzmann Method (LBM) code. Through serial optimization techniques such as loop fusion and pointer swapping, I significantly enhanced the execution efficiency of the code. Moreover, by modifying the data structures and adopting memory alignment techniques, I paved the way for compiler code vectorization. Ultimately, by leveraging OpenMP for parallel acceleration, I achieved a performance enhancement from a single core to 28 cores, substantially improving the program's performance and efficiency.