AHPC Assignment 1: Performing Serial Optimisations on a Lattice-Boltzmann program and running it in parallel using OpenMP

Marc Goulding - mg17752

A sample program for the Lattice-Boltzmann Method algorithm (LBM) is optimised to run in parallel on a BlueCrystal Phase 4 supercomputer 525 Lenovo nx360 m5 compute node, containing 2 Xeon Intel E5-2680 v4 Broadwell CPUs. First serial optimisations are performed, followed by compiler directives that improve the use of vectorised SIMD instructions. Finally, OpenMP compiler directives are used to achieve a performance *23.42x* faster than the original program. The result is a program that runs in *0.78s* instead of the original *31.8s* for a *128x128* grid. A scalability analysis of the program from 1 core up to 28 cores is performed along with a roofline model to compare the program's performance to the CPU's peak performance.

1. Serial Optimisations

Before running the algorithm on multiple cores in parallel, serial optimisations are performed on a single core to reduce the processor's total workload.

1.1 Compiler and Flags

Helping the compiler produce a faster build is the most straightforward way of optimising a program's performance. This is done using the latest version of a compiler along with compiler flags that help with optimisation. On BlueCrystal Phase 4, the default compiler is the GNU's *GCC 4.8.5* compiler. However, using the latest 2020 version of Intel's ICC compiler produces faster binary. Changing compilers from *GCC 4.8.5* to *ICC 19.1.3* gave a speed-up of *1.21x*. Intel promises faster runtimes with its LLVM-based C/C++ 2021.1 compiler, but this claim didn't hold experimentally. The ICC compiler resulted to be a faster alternative.

The ICC compiler allows for the `-O3` optimisation flag to optimise the resulting binary for performance, but using a more aggressive optimisation flag such as `-Ofast` contains multiple optimisation flags at the same time and produces even faster binary. It was also noticed that the target processor supports Intel AVX2 SIMD instructions, which allow for vectorising loops where the same instruction is applied to multiple data points. To inform the compiler that these instructions are available, the `-xAVX2` flag was added. It was later replaced by `-xHost`, since it instructs the compiler to generate SIMD instructions based on the target CPU architecture. The flag `-fast` includes `-Ofast`, `-xHost`, `-ipo`, and `-static`, so it was used instead to include all these flags. This achieved a *1.4x* speed-up.

After searching through all flags available, `-finline-functions` was added as a compiler flag because it attempts to in-line functions, trading size of the executable binary for speed. A 1.09x speed-up was recorded. Also, the flag `-m64` informs the compiler to compile for a 64-bit data model, but `-xHost` already accounts for this, so no speed-up was noted when adding this flag.

1.2 Loop Fusion

Performing a roofline analysis of the program indicates that the algorithm is memory-bandwidth bound, meaning that memory transfers are the main bottleneck in performance. As a result,

improving data footprint and movement, and cache access patterns will greatly improve performance.

GNU's profiler showed that the time-critical function in the program is `timestep()`, which in turn calls the `propagate()`, `rebound()`, `collision()` and `av_velocity()` functions. The reason why these functions take the longest to run is because all four functions each iterate through the grid independently. These loops can be merged into one to only iterate through the grid once and perform all computations together. This would greatly reduce the number of memory accesses and improve memory bandwidth efficiency. As a result, all the code is contained within the `timestep()` function, which returns the average velocity for that iteration of the algorithm. Instead of one memory access nested loop per function, there is only one nested loop for all functions. This improved runtime by *1.18x*.

1.3 Arithmetic Simplification

The compiler is able to simplify many of the arithmetic expressions in a program. However, explicitly simplifying the arithmetic operations by applying basic mathematical principles ensures that these optimisations take place and did give a minor speed-up that increases with the number of iterations performed by the algorithm.

The compiler flag `-no-prec-div` optimises divisions by changing them into multiplications by the reciprocal of the denominator (i.e. *A/B* becomes *A * 1/B*). However, I believe it is good practice for one to simplify his own mathematical expressions before using them, so I performed this optimisation on the code directly instead of relying on the compiler.

1.4 Pointer Swapping

In the original program, the entire grid data structure or array is copied from `cells` to `tmp_cells` and back on each loop iteration, creating a huge unnecessary memory bandwidth inefficiency. This was replaced with only one transfer from `cells` to `tmp_cells` in a loop iteration, followed by a pointer swap between `tmp_cells` and `cells`, making the results of the iteration stored in `tmp_cells` now be stored in `cells`, allowing the program to repeat the iteration in the next step without having to transfer all the data from one pointer address to another.

2. Vectorisation Optimisations

Vectorisation allows for instructions that run on multiple data to be reduced to single-input multiple-data (SIMD) instructions. This means that for the same amount of computation, less instructions are needed, since only one instruction is required for multiple data instead of multiple instructions. This greatly improves performance as the total workload on the processor is reduced and loop overhead is reduced. Vectorisation is a great way to efficiently drive many ALUs with a minimal set of instructions..

To help the compiler perform vectorisation optimisations, first a vectorisation report was created using the `-qopt-report=5` compiler flag. The `-xHost` compiler flag was already informing the compiler that SIMD instructions were available for the target Intel processor, but the report indicated that many vectorisation opportunities were being missed and memory accesses were unaligned with the caches, creating cache misses and inefficiencies in memory bandwidth.

2.1 Data Structure Transformation

The first step in helping the compiler vectorise was to change the main data structures `cells` and `tmp_cells` from an array of structures (AoS) to a structure of arrays (SoA). This way the compiler can identify vectorisation opportunities across the arrays, whereas before it

couldn't identify these across the structs. The speed up was of *1.87x*.

Another simple data transformation was to swap `double` data types for the `float` data type, improving memory bandwidth by a factor of 2. It was also noted that the `obstacles` data array only contained boolean values of `1` or `0`, but it was being stored in 32-bit integers. This was replaced for the 8-bit wide `char` data type, making the `obstacles` array occupy 4 times less memory than before. This increases the number of array values loaded into cache on each read. As a result, the processor needs to perform less reads from memory to load the same amount of data into the caches, improving our memory bandwidth.

2.2 Pointer Aliasing and Cache Alignment

To further aid the compiler in detecting vectorisation opportunities, the `restrict` keyword was added to all the pointers so the compiler knows there is no aliasing or overlapping between them in the inner time-critical loop. The keyword `const` was also added wherever possible to inform the compiler that these variables won't change.

Finally, memory alignment techniques were used to maximise cache efficiency. More data is used on every cache load, reducing the number of cache loads required and number of cache misses. By swapping `malloc()` and `free()` calls to `_mm_malloc()` and `_mm_free()` calls, data storage becomes aligned with the caches to a requested boundary, which was set to 64 bytes since cache lines for the target processor are 64 bytes wide. For static memory, `__declspec(align(64))` was used for cache alignment. Then, `__assume_aligned()` was used to tell the compiler the memory was aligned before time-critical loops.

To ensure the compiler vectorises the intended loops, the compiler directive `#pragma omp simd` was added before all the target loops to vectorise. This explicitly forces the compiler to vectorise these loops if possible. The `-qopenmp` compiler flag is necessary to do this.

As a result of all these optimisations, the vectorisation report stated that all loops intended were vectorised correctly, obtaining a huge speed-up of *2.8x*.

3. OpenMP Parallelism

OpenMP is a heterogeneous programming shared-memory fork-join interface API that allows for both data and task parallelism across multiple threads within the same node.

3.1 Parallel Compiler Directives

OpenMP is implemented as compiler directives. This means that by simply adding `#pragma omp parallel for` before a for loop, the program forks into multiple threads where each thread runs one iteration of the loop. The number of threads to use is defined by the environment variable `OMP_NUM_THREADS=28`. Only outer loops were parallelised since inner loops are being vectorised. Adding this before every time-critical loop gives a huge speed-up of *3.29x*.

3.2 NUMA-Aware Initialisation

In a shared-memory paradigm such as OpenMP, any core can access memory written by another core. However, accessing memory from another socket is slower than accessing memory from the same socket, since the data needs to be access through the socket interconnects and the other socket's memory controllers are used. To reduce remote memory accesses between NUMA (Non-Uniform Memory Access) regions, data initialisation is parallelised in the same way as its access loops. Memory is then allocated and initialised in the same NUMA region as the cores that access it.

NUMA-aware initialisation only helps if

the threads remain in the same NUMA region throughout the execution of the program. This is known as *thread-pinning* and is achieved by setting the OpenMP environment variables `OMP_PLACES=core` and `OMP_PROC_BIND=true`. This pins each thread to the core it was first assigned to.

3.3 Parallelised reduction

The LBM algorithm contains two addition reductions in the collision() computation which can be run in parallel to improve performance by *1.15x*. This was done by adding the `reduction(+:tot_cells, tot_u)` clause to the `#pragma omp parallel for` compiler directive. This avoids unnecessary expensive thread synchronisations when running the loop in parallel.

4. Scaling and Roofline Model

The graph below displays how performance improves as more cores are used by the program.

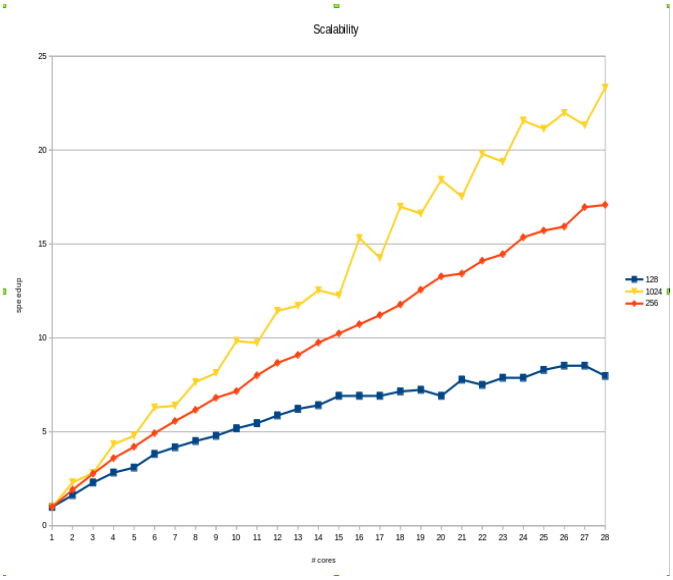

Figure 1: Scalability from 1 to 28 cores

The yellow, orange and blue lines correspond to the speed-ups obtained as we increase the number of cores used by OpenMP from 1 to 28 for the *128x128*, *256x256* and *1024x1024* grids respectively. It is noted that when going from 14 to 15 cores there is a jump in performance. This is because when using 15 cores or more, two sockets are used and memory bandwidth is therefore increased (each socket has its own L3 cache). For the bigger sized problems we see perfect scaling. The data is split across more and more cores allowing it to fit more data into the L1 and L2 caches. For the smaller *128x128* problem size there is a sub-linear plateau. The data size is already small enough to it into the caches to begin with, so adding more cores doesn't make such a big difference. As the number of cores increases, we eventually use enough cores to maximise memory-bandwidth. From then on using more cores doesn't increase performance because bandwidth is our bottleneck. This is observed also in the roofline model, where the *128x128* solution achieves a high percentage of it's peak performance, leaving little room for performance improvements due to it's memory-bandwidth bottleneck. It was not possible to include a graph of the roofline model due to errors with BlueCrystal Phase 4 and the X-forwarding functionality when using ssh.

It is also noted in the graph that using OpenMP gives *7.9x, 17.08x* and *23.32x* speed-ups compared to the serial versions of the program. Performance increases with problem size because

5. Conclusion

Serial optimisations combined with vectorisation methods greatly reduced the runtime of the Lattice-Boltzmann Method program. The final runtimes were *0.78s, 2.75s* and *10.90s* for the *128x128*, *256x256* and *1024x1024* problem sizes respectively. A *41.02x* improvement when compared to the original runtimes. OpenMP paralellisation introduces the possibility of running algorithms that otherwise might be too expensive to execute, and will continue to unlock possibilities in High Performance Computing in the future as supercomputers grow in size and become more heterogeneous in design.