High Performance Computing Coursework Report – Philip Mortimer [ia21629] (2100451)

Introduction

I have produced an optimised serial implementation of the lattice Boltzmann (LBM) code and effectively vectorised the code. I have then parallelised the code using OpenMP, enabling it to be run across all 28 cores of a single node on BlueCrystal, the University's supercomputer. My serial, vectorised and parallelised implementations all run significantly faster than the provided ballpark times, as outlined in *Table 2*.

Serial Optimisation

Initial Compiler Selection

Having been provided with a base serial LBM implementation, I experimented with a number of compiler configurations to determine which was fastest. I compared the latest versions of the two main compilers available to me: the GNU compiler (GCC) and the Intel compiler (ICC). I found that ICC performed better than GCC, likely because the Intel compiler is optimised to produce performant code for its own chipsets; BlueCrystal uses Intel E5-2680 v4 CPUs.

In order to avoid premature compiler optimisation, I tested all serial improvements using ICC and '-Ofast' on the 128 by 128 grid. This establishes a baseline time of 25.0 seconds. Upon completion of the serial code, I further experimented with compiler flags to determine the optimal configuration for the code.

Loop Fusion

For each timestep within LBM, 5 key functions are called: *accelerate_flow*, *propagate*, *rebound*, *collision* and *av_velocity*. Each function is called once per timestep, with all of them bar *accelerate_flow* iterating through every element in the grid and thus running in $O(xy)$ where x and y are the grid dimensions. *accelerate_flow* runs in $O(x)$. In order to save unnecessary iteration, we can fuse the main loops from each function, excluding *accelerate_flow*, into a single loop. To achieve this, the cell elements from the previous round are treated as read-only, and the updated state is written to the *temp_cells* grid. A pointer swap is then used to treat this new grid as the next state. This change reduces the runtime from 25.0 seconds to 22.8 seconds.

Arithmetic Optimisation

A speedup can be achieved by optimising the arithmetic calculations within the *collision* segment. For instance, $d[1] = w1 * localDensity * (1 + \frac{u[1]}{cSq} + \frac{u[1]*u[1]}{2cSq*cSq} - \frac{uSq}{2cSq})$,where uSq, cSq and $w1$ are all constants, can be rewritten as $d[1] = \frac{w1}{2cSq^2} * localDensity * (2cSq^2 + 2cSq * u_x + u_x^2 - cSq * uSq)$. Note that the constant terms can be computed before entering the main loop (e.g. $c_1 = \frac{w1}{2cSq^2}$) which saves repeated calculation. Importantly, it allows for floating point divisions to be replaced with floating point multiplications, which are significantly faster. By applying similar logic for all terms and by computing terms common to all components (such as $2cSq^2 - cSq * uSq$) once, we can save additional computation and improve performance. This leads to the LBM code running in 21.8 seconds – a 1.0 second improvement.

Compiler Optimisation

I experimented with a range of compiler flags and determined that for ICC, the following flags performed best "-Ofast -xHost -ipo". '-Ofast' enables a number of aggressive compiler optimisations

and is a level above '-O3'. '-xHost' generates code that is tailored to the instruction set used on BlueCrystal, which improves performance on the supercomputer at the expense of portability. '-ipo' enables inlining of functions called from other files and is part of a suite of options typically enabled with the '-fast' flag. This compiler setup leads to a runtime of 16.0 seconds. This significant speed up over just '-Ofast' is predominantly down to the '-xHost' flag, showing that ISA (instruction set architecture) specific gains are substantial.

When compiling with GCC, I concluded that the following flags worked best: "-Ofast -march=native". These flags ensure that the compiled code is tailored to BlueCrystal's architecture and that aggressive - and potentially unsafe – optimisations occur. This setup achieves a runtime of 19.2 seconds.

Code Vectorisation

Conversion to Structure of Arrays

In order to vectorise the serial code, the array of structures storing the grid cells is replaced by a structure of arrays. The restrict keyword is used to indicate that pointers don't overlap and hence that vectorisation can safely occur. Making this change improves runtime from 16.0 seconds to 6.8 seconds. The reason that this improves performance so dramatically is that it allows BlueCrystal to generate SIMD instructions which allow for the same instruction to be executed on multiple data items simultaneously on a single core (natural parallelism). As our main loop is vectorised, we see a remarkable performance boost. Stencil code like LBM is typically memory bandwidth bound and thus vectorisation helps improve performance in this area. This is because each cache load with a structure of arrays loads data useful to the computation.

Data Alignment and Loop Counter Assumptions

In order to aid vectorisation, I have aligned the grid items in memory to a specified byte boundary, using *_mm_malloc(ptr,bytes)*, *_mm_free(ptr)* and *__assume_aligned(ptr,bytes)*. Data alignment improves performance as data is aligned along cache line boundaries. Consequently, aligned data leads to fewer data loads (i.e. maximum cache line utilisation).

In addition to data alignment, providing information about loop counters can help to optimise code. As LBM's main loop depends on the dimensions of the grid (x x y), specifying the divisibility of these elements can help with complication. In this case, the compiler can safely assume that x and y are both divisible by 128. This optimisation does come at the expense of code generalisability (i.e. the code may not work when x and y are not divisible by 128), but the speedup provided is substantial and thus is a worthwhile change.

Generally, aligning data to cache line boundaries extracts optimum performance. Hence, 64 bytes (or some multiple of 64) is likely to work best. In order to test this hypothesis, I tested my code's runtime using a range of values.

Alignment (bytes)	**Runtime (s)**			
	128 x 128 Grid	128 x 256 Grid	256 x 256 Grid	1024 x 1024 Grid
32	5.7	13.0	48.3	259.4
64	5.5	12.8	48.5	258.5
128	5.4	13.0	48.6	261.7
256	5.3	12.9	48.5	261.2
512	5.3	13.3	48.6	256.5
1024	4.8	12.8	47.9	286.6
2048	5.0	13.1	48.1	289.3

Table 1 - Performance with different alignment configurations. Green cells represent the best runtime for that grid size and red represents the worst runtime. In case of joint fastest / slowest times both cells are highlighted.

Table 1 demonstrates that 1024-byte alignment marginally outperforms other alignment options, except in the case of the 1024 x 1024 grid. In order to achieve best performance across a range of grid configurations, the LBM code dynamically selects alignment size based on the number of grid elements.

OpenMP Parallelisation

Parallelisation

'#pragma omp parallel for' directives are used to run the core loop across multiple threads. In order to maximise performance, the loop iterating through all grid cells has been manually collapsed, leading to a 0.3 second speed up. It is interesting to note that utilising OpenMP's collapse function to automatically collapse the two loops lead to a much slower program. This is likely because my manually collapsed variant computes the corresponding ii and jj coordinates more intelligently- with $ii = i \,\%\, x$ and $jj = (ind - ii) \gg \log_2 x$. Flattening the loop means that loop peeling is no longer performed. Implementing loop peeling manually for the flattened loop degrades performance as the arithmetic overhead and reduced parallelisation outweigh the computation time saved by not having grid indices wrap around.

The LBM code now divides the loop iterations evenly amongst 28 threads. As the average velocity is computed by summing the results across numerous iterations together, a reduction(+:tot_u) clause is inserted to allow for this computation to occur.

NUMA

BlueCrystal uses a dual-socket shared memory system. In our parallel system, if a CPU is accessing memory on the other socket, memory read time almost doubles. In order to ensure that data is stored in the closer socket where possible, data is initialised in a '#pragma parallel for' loop, similar to that of the main timestep loop. This maximises the likelihood that the cell elements will be stored in the closer socket for that thread, improving access time when it comes to performing the LBM computation.

OpenMP allows for control of how threads are pinned to cores using the OMP_PROC_BIND variable. I found that OMP_PROC_BIND=true produced the best results, although I found that as long as OMP_PROP_BIND was not set to 'master', most choices seemed to have no noticeable effect.

Performance Scaling

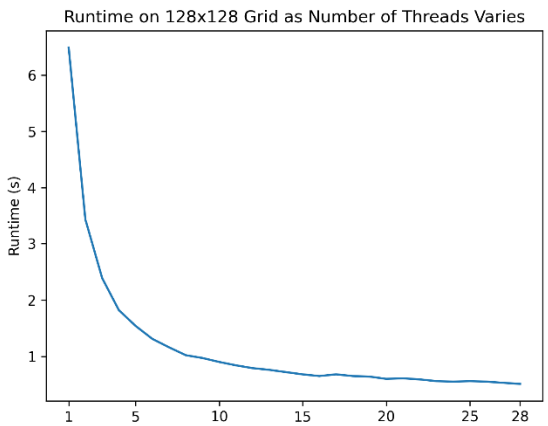

Figure 1 - Runtime scaling with number of threads.

As the number of available threads increases, the runtime decreases. Initially, doubling the number of threads doubles the performance. However, as the number of threads increase, this effect diminishes. This performance boost occurs because doubling the threads halves the amount of work that each one does. As they execute in parallel, this effectively halves the runtime. However, with 28 threads, for instance, each thread only has to process 585 cells. At this point, the overhead of thread creation, in combination with the increased accessing of NUMA

memory from the more expensive socket, means that each additional thread offers less runtime improvement.

Roofline Analysis

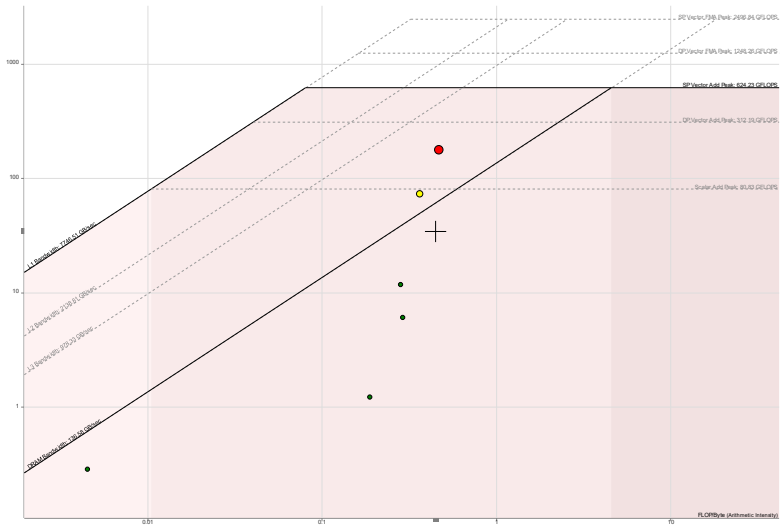

Figure 3 - Roofline analysis of LBM on 128 x 128 grid.

In *Figure* 3, the red dot in the roofline analysis represents the performance of the main loop within the timestep function. The LBM code sits between the DRAM and L3 cache roofs, demonstrating that the program is not bound by DRAM bandwidth and is likely (although not certainly) bound by L3 cache bandwidth. For all roofline peak bandwidths other than L1 cache, the LBM code would appear to be memory bandwidth bound. Given that stencil programs are almost always memory bandwidth bound, I thus conclude that the LBM code is memory bandwidth bound.

Using a single thread, the LBM program achieves 39% peak memory bandwidth utilisation relative to the STREAM benchmark, which indicates that the program has been optimised relatively well in this regard even if there is still room for improvement.

Final Performance

Grid Size	Serial Runtime (s)	Vectorised Runtime (s)	Parallel Runtime (s)
128 x 128	16.0	4.8	0.5
128 x 256	32.2	12.8	0.8
256 x 256	129.1	47.9	2.6
1024 x 1024	537.8	256.5	11.5

Table 2 - Final LBM runtime.

Conclusion

I have produced a strong LBM system that performs significantly better than provided benchmark times. I have achieved this through optimising the provided serial code by applying loop fusion, arithmetic optimisations and by selecting a suitable compiler. The code has then been vectorised to allow for natural parallelisation of the code using SIMD instructions. Finally, the code has been parallelised using OpenMP to allow it to run on 28 threads in parallel.