HPC Coursework

Luca Gough

January 2024

1 Introduction

This project is an analysis of performance optimisations for a structured grid program specifically a D29Q lattice Boltzmann implementation modelling fluid flow around obstacles. The first three sections describe the serial, vectorised and parallel optimisations made to the code and the final section focuses on experiments and analysis.

2 Serial Optimisation

This sections explains optimisations made to the code and compiler that improve it's performance on a single core. The final serial execution time for the 128x128 input was 20.2s.

2.1 Compiler

2.1.1 Compiler Choice

The choice of compiler greatly affected the total runtime of the program because different compilers are able to optimise the code in different ways. Intel's ICC compiler (Blue Crystal Phase 4's compute nodes use 'Intel Xeon E5-2680 v4' CPUs), written specifically for the x86-64 instruction set architecture should be able to optimise more aggressively than the GNU Compiler Collection (GCC) as it can make more assumptions about the hardware. For example ICC can insert Advanced Vector Extension (AVX) instructions which take advantage of 256 bit special purpose SIMD registers, capable of storing 8, 32 bit floats.

2.1.2 Compiler Flags

Compiler flags enable the user to toggle certain features of the compiler. Specifically they can control exactly what optimisations the compiler is allowed to perform. After some experimentation the best performing combination of flags was '-Ofast -mtune=native -xCORE-AVX2'. 'Ofast' toggles all compiler optimisations as well as a number of flags that disable error handling for math routines and allow less accurate floating point operations. '-xCORE-AVX2' specifies the target architecture allowing it to perform hardware specific optimisations.

2.2 Loop Fusion

Loop fusion describes the process of merging the bodies of multiple loops such that the iteration only happens once. In Lattice Boltzmann this corresponds to computing the 'propagate', 'rebound' and 'collision' steps in a single loop. Iterating over the whole grid once allows the compiler to reduce the number of memory loads/writes to once per grid cell rather than three times (Once in each step). This had the affect of reducing the overall runtime by around 3 (128x128 input) seconds. The loop fusion also eliminates the need to copy the values from the scratch space into the main grid further reducing memory access.

2.3 Branch Removal

Modern pipelining architecture has significantly reduced the time required to execute a group of instructions, allowing the CPU to compute an instruction on each clock cycle. Branching presents a challenge for pipelining as the CPU faces uncertainty about which branch path to retrieve instructions from. In an effort to combat this delay most processors now integrate branch prediction hardware however a misprediction now requires the entire pipeline to be cleared and re-fetched. Therefore it is crucial to avoid conditional branches wherever possible. In Lattice Boltzmann's main iteration of the grid it has different logic for *obstacle* cells and *normal* cells. This is handled with an 'if statement' at the start of the loop (Listing 1), if the condition could be evaluated at compile time the compiler could potentially remove the branch entirely however in this case it must be evaluated dynamically. We can remove this branch manually by taking advantage of the asymmetry present in the grid. There is a much greater number of *normal* cells than there are *obstacle* cells meaning we can perform a full iteration of the grid applying the *normal* cell logic to all cells. Subsequently we can iterate

over only the *obstacle* cells and perform overwrite them with the *obstacle* logic.

```
for (int ii = 0; ii < grid_height; ii++) {
    for (int jj = 0; jj < grid_width; jj++) {
        if (!obstacle[ii + jj*grid_width]) {
            // Normal cell logic...
        } else {
            // Obstacle cell logic...
        }
    }
}
```

Listing 1: **Main Loop with conditional branch:** The code selects different logic depending on the cell type.

```
for (int ii = 0; ii < grid_height; ii++) {
    for (int jj = 0; jj < grid_width; jj++) {
        // Normal cell logic...
    }
}
for (int ii = 0; ii < obstacle_count; ii++) {
    obstacle_index = obstacles[ii];
    // Obstacle cell logic..
}
```

Listing 2: **Main Loop without conditional branch:** The code iterates over the obstacle cells twice.

This change made a small improvement (around 0.5s for the 128x128 input) to the execution time however, it will have more of an affect when the code is vectorised.

2.4 Loop Invariant Hoisting

Profiling the code with Intel Vtune highlighted that the iteration of the grid significantly contributes to the overall runtime. Given that Lattice Boltzmann is memory bandwidth bound reducing the memory access and floating point operations inside the loop should have the biggest effect on runtime. The start of the loop body computes the index of each cells neighbors, this requires two 'if statements' and a few integer operations. Crucially the computed values are invariant throughout the program, meaning they can be pre-computed and loaded from memory when they are required in the loop. This optimisation removes the 'if statements' preventing any chance of branch misprediction. This reduced the runtime by a further 3 seconds.

3 Vectorisation

Vectorisation leverages special purpose hardware built into modern processors to perform a single instruction on multiple data elements simultaneously (SIMD). Large 128-256 bit registers can store 4-8 floating point values on the CPU and SIMD arithmetic hardware can perform operations on those values in parallel. OpenMP provides a set of 'pragmas' for invoking SIMD code generation which will generate SIMD instructions at compile time. Xeon Broadwell's AVX registers can store a 256 bit operand or 8 bytes so vectorizing the inner loop of the grid iteration should yield an 8x performance increase on each floating point operation. In reality this is not the case because data used in consecutive loop iterations is not consecutive in memory, this means we have to perform 8 separate loads to populate our SIMD register. Loads from main memory may take around 100 cycles (8 consecutive loads may not individually take 100 cycles due to interleaved memory access), a SIMD instruction will take around 1-3 CPU cycles. This implies that in order for SIMD to be effective we must first optimise our memory layout to minimise unnecessary loads because memory bandwidth is our bottleneck. This is achieved by ensuring that the data required for each operation is stored contiguous in memory. We can implement this in a Structure-of-Arrays format.

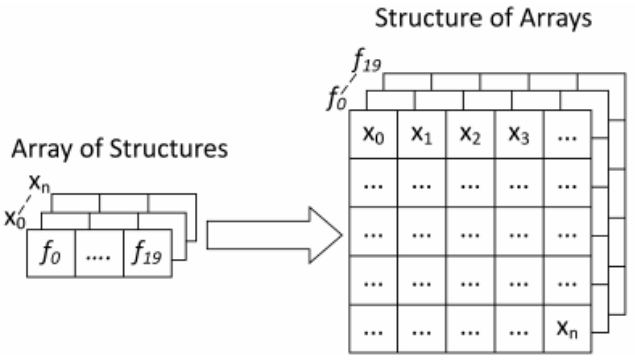

Figure 1: **Array-of-Structure vs Structure-of-Arrays:** Left shows an array of structures format where each grid cell has it's own struct containing it's relevant information. Right shows a structure of arrays where a there is a single array for each field in the struct.

Utilising a structure-of-arrays means that each SIMD operation can load it's 8 operands in a single instruction. This can be further optimised by aligning the start of each array to 64 byte boundary (Blue Crystal Phase 4's L1 cache line size is 64 bytes), this ensures that a floating point value cannot be allocated across the boundary between two cache lines requiring a load of both lines (Cache cannot address individual bytes).

After all the serial and vectorised optimisations the execution time for the 128x128 input was 6.8 seconds.

4 Parallelisation

Since Lattice Boltzmann is a structured grid problem [2] there is no loop dependency between grid cells for a given iteration of the grid. This means we can easily compute the iteration of each cell or of groups of cells in parallel. However there is a temporal dependency meaning we must synchronise the computation of each cell at the end of each time step. We can use Amdahl's law [1] to calculate the theoretical speedup if we had infinite processors.

$$p = \text{Proportion of runtime that is parallelisable} \quad (1)$$

$$S = \text{Theoretical Maximum Speedup} \quad (2)$$

$$\lim_{x \to \infty} S = \frac{1}{1-p} \quad (3)$$

$$S_n = \frac{1}{(1-p) + \frac{p}{n}} \quad (4)$$

With all the previous serial and vectorisation optimisations the proportion of runtime that is parallelisable is 0.995 meaning our maximum theoretical speedup is around 230x. On 28 cores this corresponds to a speed up of 24.7x.
After extensive testing the most effective implementation was to parallelise the iteration of the rows and to vectorise the iteration of the columns of the grid. Collapsing both loops proved to be ineffective because it resulted in more jobs per thread and therefore each thread has to change jobs more frequently which incurs a significant overhead. The branch removal described in section 2.3 also resulted in a large increase in parallel performance because it ensured that the logic in the loop body was invariant across all iterations. This means that the workload in each job is equally sized, so each thread should finish at the same time, minimising threads waiting to synchronise at the end of the iteration. In addition to this it also proved effective to use OpenMP's *active* thread wait policy this causes threads to spin wait, reducing latency as they access new jobs.

4.1 Non-Uniform Memory Access

Blue Crystal Phase 4's Broadwell CPU's have two, fourteen core CPU sockets connected via a high speed interconnect, each socket is connected to a separate memory pool. However each socket can access both pools of memory via the interconnect. Communication over the interconnect adds some overhead to memory access if a socket is accessing memory belonging to the other socket. Effectively the system behaves as a shared memory model with non-uniform memory access times. Intel's memory latency checker reports 65ns for local memory access and 125ns for remote memory access. It is therefore crucial to minimise remote memory accesses, this is achieved by taking advantage of the first touch policy, which dictates that the processor will only allocate memory when it's is first accessed. A thread will allocate memory in it's closest available RAM. Therefore, we can minimise communication over the interconnect by forcing threads to initialise the memory locations that they will eventually use (and by pinning those threads a particular core). In practice there will still be some communication as each core needs to maintain cache coherency with the cores in the other socket. However the NUMA aware version of lattice Boltzmann achieves a speedup of 1.3x.
Running across all 28 cores, with all the previous serial and vectorised performance optimisations, the final execution time on the 128x128 grid was 0.84s and 18.3s for the 1024x1024 grid.

5 Performance Analysis

5.1 Roofline Analysis

The roofline model [3] provides a framework for quantitative analysis of a program's performance with respect to the hardware's theoretical peak. It plots a program's operational performance against it's operational intensity. Operational intensity is defined as the ratio of the program's operational performance to it's memory traffic. We can then include lines representing the peak memory bandwidth and peak operational performance of the hardware. Since Blue Crystal Phase 4 has 3 levels of cache with increasing access bandwidth each level has it's own memory bandwidth roofline. Similarly SIMD operations provide an increased number of FLOP/s than scalar operations and therefore raise the operational performance roofline. Figure 2 shows the roofline model applied before and after the branch removal optimisation described in section 2.3.

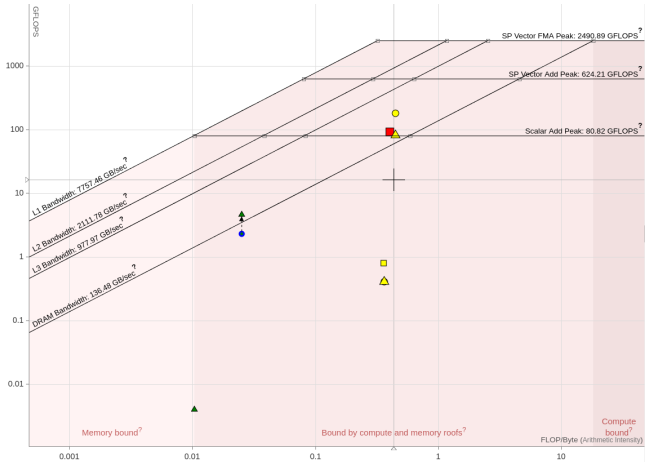

Figure 2: **Roofline Comparison:** This chart shows the roofline model applied to the Lattice Boltzmann code before and after the branch removal optimisation 2.3. Square's are loops before optimisation and circles are loops after optimisation. The triangles represent the **non**-NUMA aware, optimised code. The colour represents the percentage of the total execution time (Red: $\% \geq 20$, Yellow: $20 > \% \geq 1$, Green: $1 > \%$).

The two highest markers represent the main loops in both versions, both have roughly the same operational intensity (0.405 FLOP/Byte). However the optimised version has around 2x the FLOP performance, this is because removing the branch allows the compiler to vectorise the loop, enabling it to perform more FLOP/s for the same quantity of data fetched. It's clear to see from the graph that both implementations are not bound by the DRAM bandwidth roofline implying that they both make use of the higher bandwidth caches. However the optimised program still only uses 1/10th the bandwidth of L1 cache with respect to the peak single precision SIMD FMA (Fused Multiply-Add) FLOPS. This could be improved by reducing the job size allocated to each thread, allowing them to fit more relevant data in the faster cache levels. Another interesting result present in the roofline graph is the effect of NUMA. The NUMA-aware version of the code is able to perform 2.2x the number of FLOP/s, this occurs because each thread is no longer delayed by slower remote memory accesses.

5.2 Scaling Analysis

As we increase the number of threads we expect to see linear scaling for the first few cores as the threading overhead and thread synchronisation delay is negligible. However we can expect this to plateau to sub-linear scaling as these delays become more significant in addition to this Amdahl's law tells us that serial sections of code become performance bottlenecks. This is consistent with the data presented in figure 3. The peak performance increase was 16.8x, this is significantly below our predicted upper bound of 24.7x. Intel Vtune's flame chart reports that our average simultaneous core utilisation was at 20 CPU cores and that threads were predominantly waiting in the '_kmpc_reduce' function, this implies that the delay is caused by threads having to synchronise to add their results together. A reduction is the OpenMP construct to avoid race conditions when performing operations on shared memory. This could be achieved by writing results to separate memory and collating at the end. Minimising the overhead of each thread accessing a mutex.

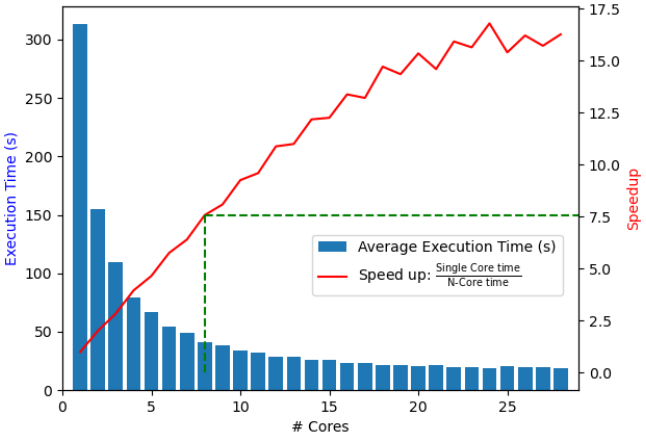

Figure 3: **Multi-Core Scaling (1024x1024 input): Blue:** Execution time on n-cores (Averaged across 5 trials). **Red:** Speedup relative to single core performance **Green:** Transition to sub-linear scaling

6 Conclusion

In summary, in order to write performant software it is critical to first understand our target hardware. This thesis is core to compiler choice, memory utilisation, data structuring and NUMA awareness. In conjunction it is imperative to quantify an upper bound relative to our hardware's peak performance. These ideas allow us to identify performance bottlenecks and employ significant optimisations, maximising utilisation of the hardware.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. 3

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, oct 2009. 3

[3] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009. 3