1 Introduction

This report will the explore the optimizations applied to the implementation of the d2q9-bgk lattice boltzmann scheme. It will initally look at the serial omtimizations and how they affected the performance of the alogirthm It will then look at the parallel implementation and how this affect the performance with different number of cores and problem sizes.

2 Serial Optimisation

2.1 Reduce Memory Accesses

In the original implementation of the algorithm for each timestep the `cells` array was looped over the 4 times, in `propagate`, `rebound`, `collision` and `av_velocity`. This resulted in repeated stores and loads of the same sections of memory. To prevent this the first 3 separate functions (those in the `timestep` function) were fuzed into a single loop. This meant the same sections of memory were used closer together making it is more likely for them to still be in cache. The function `av_velocity` was then repeating calculations that already took place in collision and requiring an additional loop over cells. The result was therefore calculated in the single parse over the cells and returned from the timestep function. Similarly in propagate and collisions the values were switched between `tmp_cells` and `cells` multiple times. In the new implementation the `tmp_cells` array was used as the "answer" space and stored only the next timesteps cell values. This then only required a single write to `tmp_cells` each timestep. At the end of the timestep the `tmp_cells` and `cells` array's pointers were then swapped which set the `cells` array to the correct value without having to write directly to the array.

2.2 Vecotrisation

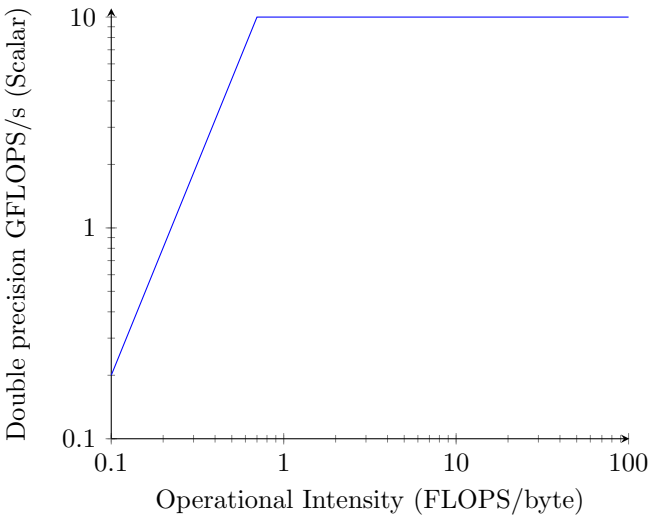
Having improved the implementation of the serial code, there is now a clear critical section of the code, inside the single pass of the cell. This is where the majority of the computation takes places and thus is where most of the time of the program is used. Since this section performs multiple mathematical computation on the input array, there is an additional approach to optimization other than parallelizing it. This is approach makes use of SIMD, (single-instruction multiple data). This is where compilers are able to use vector registers and instructions to make multiple computations on a chuck of data (a vector) with a single instruction. This has the potential for large performance gains as fewer instructions are required for the cells array.

In many cases the compiler is able to automatically vectorize code blocks. However if the data is not aligned then the compiler will have to complete a prior step, called the "prologue" step to process this unaligned data, or alternatively use unaligned instructions (which are less efficient). The compiler is also not able to assume that an array of floats, such as the arrays of speeds in the new `t_speed` struct are aligned. Therefore it will have to complete this prior step regardless. To prevent this the arrays can be aligned when they are created using the `__mm_malloc` function. This ensure the created array is aligned on the request boundry. In this case 64 bytes was used to match the size of the cache line in an *intel xeon e5-2680*. Compiler directives can then be use to tell the compiler the arrays are aligned. This allows the compiler to skip the "proluge" step and use the aligned instructions.

In the original implementation the speeds for each cell were stored in a structure (`t_speed`) and the whole grid of cells was an array of these structures. This implementation does not lend it self to vertorization as it can result in unnecissary fetches from memory. This is because when a single speed value is fetch an entire cache line (64 bits) will be used to fetch the strucutre. This results in a waste of memoery bandwidth as the rest of the cache line is not used. Therefore swithcing the implementation to use a strucutre of array (where each speed is a separate array) allowed the code to be vectorized and reduced the memory bandwidth required.

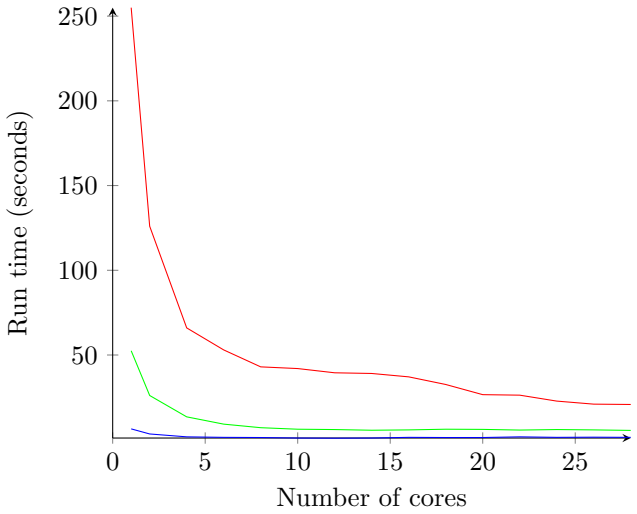
To further reduce to overhead of vectorizing `restrict` was used through the code. This tells the compiler that the pointers cannot be aliased and therefore it has to reload the values fewer times as it can be sure the value will not have changed.

```
1 cells_ptr->speed0 = _mm_malloc(params->nx * params
    ->ny * sizeof(float), 64);
2 __assume_aligned(cells->speed0, 64);
```



3 Parallel (OpenMP)

For this implementation the inner loop of the combined parse was vectorized and this allowed for the outer loop to then be parallelized.



Results			
Number of cores	128x128	256x256	1024x1024
1	6.4	52.5	255
2	3.4	26.1	126
4	1.7	13.5	66
6	1.4	9.2	53
8	1.3	7.1	43
10	1.0	6.2	42
12	0.9	6.0	39.5
14	1.0	5.6	39.1
16	1.4	5.8	37.1
18	1.3	6.2	32.6
20	1.3	6.1	26.6
22	1.7	5.7	26.3
24	1.4	6.0	22.8
26	1.5	5.8	21.0
28	1.4	5.5	20.8

tableParallel scaling results from 1-28 cores