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 Compiler options

Many modern compilers provide various options to optimize the given code.

As shown in **??** simply using these provided options with the compiler can have a profound impact on the resulting program's speed.

Results	
Compiler flag	Run time
-O0	6.4
-O1	3.4
-O2	1.7
-O3	1.4
-Ofast	1.3
-Ofast -mtune=native	1.0

Table 1: Serial run time with different compiler flags

2.2 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.

2.3 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.

```
1 // Allocate aligned data
2 cells_ptr->speed0 = _mm_malloc(params->nx * params
    ->ny * sizeof(float), 64);
3 // When using the array allow the compiler to
    assume alignment
4 __assume_aligned(cells->speed0, 64);
```

Listing 1: Example of memory allignment for a 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, as shown in Listing 1. 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.

Another useful hint to assist the compiler in its optimizations is to use the `restrict` keyword. This tells the compiler "that for the lifetime of the pointer, only the pointer itself or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points". In practical terms this reduces the number of times the pointer value has to be fetched from memory as the cached value can be used repeatedly. In a memory bound problem this can therefore be an important step in reducing the memory bandwidth used.

```
1 __assume(params.nx%128==0);
2 __assume(params.ny%128==0);
```

Listing 2: Additional compiler hints

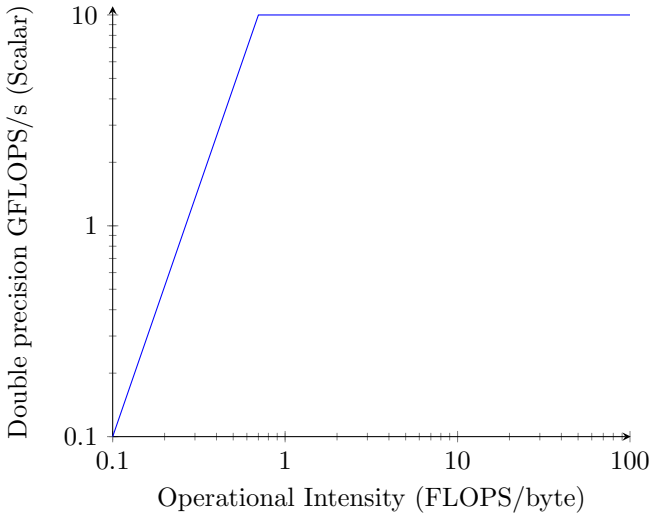
The intel compiler also allows for other hints to be provided. In this implementation hints were porivded to inform the compiler about the size of the cells array (and therefore the size of the loops). As show in the example in Listing 2, the `__assume` directive was used, to tell compiler that the size of the cells array was divisible by a given power of 2. In the implementation all powers of 2, up to a maximum of 128 were provided (as this was the maximum input size). This should further assist the compiler in its optimizations, especially when vectorizing the inner loop.

The final step to ensuring the code was vectorized was to add the compiler directive `#pragma omp simd` to the inner for loop. This explicitly tells the compiler that this section of the code should be vectorized.

Having completed the above steps only a minor improvement in performance were observed, as seen in Figure xx, despite the inner loop being successfully vectorized. From the

roofline graph in Figure xx, it is clear that the code was memory bandwidth bound. In this case the problem arose from the data layout used for the cells.

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 vetorization as it can result in unnecessary 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 structure. This results in a waste of memory bandwidth as the rest of the cache line is not used (only the single float value for that speed is actually required). Therefore switching the implementation to use a structure of arrays (where each speed is a separate array) allowed the code to be vectorized more efficiently as each fetch for a speed only fetched a single float and therefore reduced the memory bandwidth required.



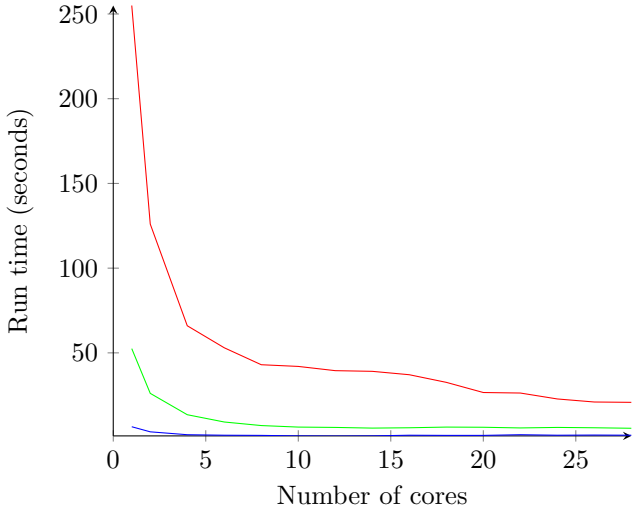
3 Parallel (OpenMP)

Having optimized the inner loop in the above section the next optimization that can be used is parallelizing. In this implementation the outer loop was parallelized while the inner loop maintained the other optimizations. This was achieved using OpenMP. This is library provided a collection of compiler directives which can be used to write parallel shared memory programs.

The first step to making the outer loop run in parallel is to use the `parallel` and `for` constructs and turn on OpenMP for the compiler with the `-fopenmp` option. The `parallel` directive tells the compiler to run the preceding for loop in parallel and `for` tells it to distribute the work evenly across the threads.

```
#pragma omp parallel for schedule(static) reduction
    (+:tot_cells) reduction(+:tot_u)
    for (int jj = 0; jj < params.ny; jj++)
    {
#pragma omp simd aligned(cells:64) aligned(tmp_cells
    :64) aligned(obstacles:64) reduction(+:tot_cells
    ) reduction(+:tot_u)
      for (int ii = 0; ii < params.nx; ii++)
      {
        ...
      }
    }
```

Listing 3: TODO



Results			
Number of cores	128x128	256x256	1024x1024
1	6.4	52.5	255.4
2	3.4	26.1	126.2
4	1.7	13.5	66.1
6	1.4	9.2	53.3
8	1.3	7.1	43.4
10	1.0	6.2	42.4
12	0.9	6.0	39.7
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

Table 2: Parallel scaling results for 1-28 cores