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.

Table 1 shows the run time of the serial optimized code (without vectorization) with different compiler flags. This shows that by simply using the `-Ofast` flag there can be a 4x decrease in run time. The `-mtune=native` option can further imporve the performance by telling the compiler to optimize the code for the specific local machine and its instruction set. The compiler flag `-no-prec-sqrt` was also used, which reduces the required precision of the square root funciton which is used in the critical section of the algorithm.

Results	
Compiler flag	Run time
-O0	111.7
-O1	34.5
-O2	34.2
-O3	31.9
-Ofast	26.8
-Ofast -mtune=native	26.5

Table 1: Serial run time with different compiler flags

2.2 Arithmetic improvements

2.3 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.

2.4 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.

```c
// Allocate aligned data
cells_ptr->speed0 = _mm_malloc(params->nx * params->ny * sizeof(float), 64);
// When using the array allow the compiler to assume alignment
__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.

```c
__assume(params.nx%128==0);
__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.

When comparing the roofline results in Figure 1 we can see a large improvement in the performance, from 2.49 GLOPs to 13.2 GLOPs, once the vectorization is introduced. Simiarly we also see a reduction in the run times as shown in Figure xx. This is as a result of fewer instructions being required to compute the result values as a single vector instruction can compute multiple values. Moreover we see the operational intensity increase, and therefore the problem become more compute bound. This is as a result of switching from an array of structures to a structure of arrays which reduces the memory bandwidth used as large section of the cache line are not wasted when fetching a whole structure which contains 9 speeds, when only 1 is required.

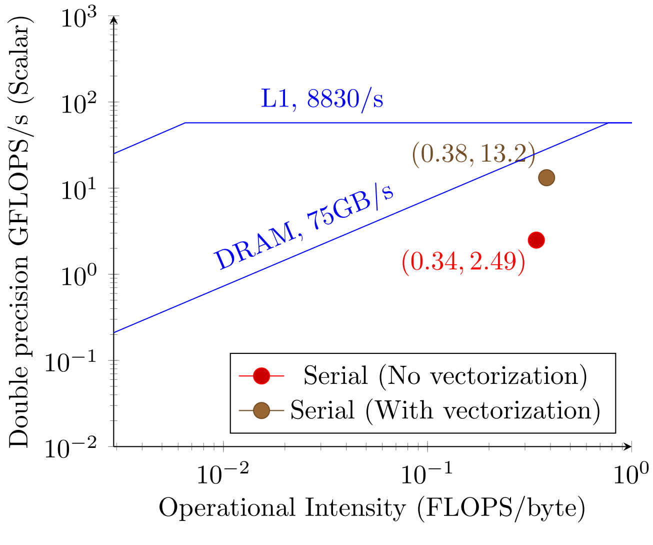

Figure 1: Roofline analysis for serial implementations

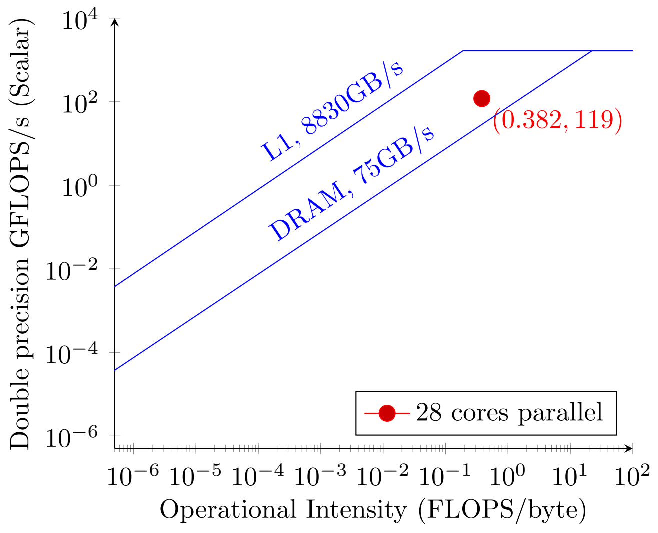

Figure 2: Roofline analysis for parallel implementations

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 library provides 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` flag. The `parallel` directive tells the compiler to run the preceding block in parallel and `for` tells it to distribute the work of the following for loop evenly across the threads.
It is then important to handle the memory sharing that takes place in the loop. Since the `av_velocity` is also calucated in this loop each loop is required to calculate the total number of cells without obstacles and the sum of the magnitude of the velocity for these cells. This requies a shared sum across all the threads. In order to efficiently calculate these shared sums a `reduction` is required. The `reduction` directive tells the compiler that the given varible is going to be accumulated over the duration of the loop. This allows each thread to calulate is own sum without having to syncronise the global shared value on every write. Once the forked section is complete and the threads syncronise then all the sums can be combined to determine the final value of the accumulated variables.
Another consideration when writing a parallel program is the number of cores to use. In a bluecrystal node there are 2 sockets, which house 2 *intel xeon e5-2680* CPUs. Each of these CPUs have 14 cores, giving a total of 28 cores per node. Table 3 and Figure 3 show the run times of the parallel code on different numbers of cores. In the larger problems (1024x1024) it is clear that as the number of cores increase the run time decreases. Initially, between 1 and 4 cores the rate of decrease is linear, meaning as the number of cores increases the run time is halved. This is as a result of the work in the critical section being shared over a larger pool of cores and thus the total time of the forked region is decreased.

However this linear relationship becomes a sublinear plateaus as the number of cores increases. The reason for this can be observed in Figure 2 which shows the roofline analysis for the parallel alogirthm running on 28 cores. The results lie in the memory bound region, this is as a result of the memory bandwidth becoming saturated as the number of cores are increased and thus the computation rate increases at a faster rate than the memory bandwidth available.
However in the smaller problems there is a reduction in run time around 14 cores. This is observed as once the core count is above 14, two sockets are in use (both CPUs) and therefore memory sharing between the sockets is required. This is where a shared section of memory is in the cache of one the other socket so in order to access that section of memory a request has to be routed through the other socket. This takes approximately two times as long as fetching from the socket's local cache. For this reason when running the algorithm on smaller grid sizes (128x128) only 14 cores were used.
In general this non-uniform memory access, NUMA, can be mitigated by pinning the threads to a specific core. Setting `OMP_PROC_BIND` to `close` tells OpenMP to bind the threads to specific cores and also as the threads are assigned, ensure they are assigned to the threads closest to the master (main) thread. Also assigning `OMP_PLACES` to `cores` ensures that each thread is used as a single place (and no thread sharing occurs such as hyper-threading). Although the threads are now pinned to individual cores this does not mean that the NUMA problem is reduced. This is because during the memory assignment, in the `initialise` function, the initial values for the cells array are assigned serially (on a single thread). This means all the values will in exist the same cache, linked to this single NUMA region. This can be prevented by ensuring the initialisation of the values is parallelised in the same way as the it is accessed in the critical region. In this case this means also using the `#pragma omp parallel for` directive when assigning the initial values to the cells array. This will ensure the data is allocated in the same NUMA region to which it is going to be required in the critical loop. The combination of these factors should therefore reduce the number of socket-to-socket memory requests allowing for the average time for each memory access to be quicker.

```
#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

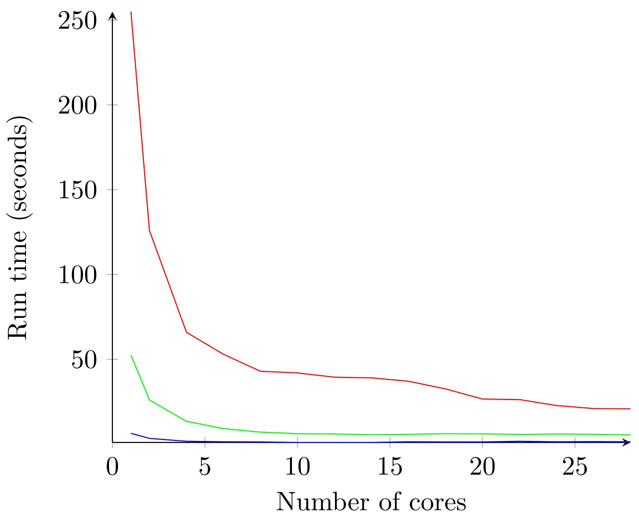

Figure 3: Parallel scaling results for 1-28 cores

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

Results			
Grid size	Serial	Vectorised	Parallel
128x128	27.0		1.15(14 cores)
128x256			1.85(14 cores)
256x256			5.68(28 cores)
1024x1024			20.8(28 cores)

Table 3: Parallel scaling results for 1-28 cores

4 Conclusion

Modern compilers offer many powerful options to assist in code optimisation. Often the most efficient approach to optimising a code can be to assist the compiler in its own optimisations. Simple changes such as using compiler optimisation flags can have a profound impact on the runtime of the algorithm. Similarly with the correct hints and memory aligned compilers are able to automatically vertorize a given code. This can dramatically reduce the run time by allowing a single instruction to compute multiple values and thus reduce the overall number of instructions. In this implementation, the vectorized code ran over 2.5 times faster than the optimised serial version. Once a code is well optimised in serial further improvements can be gained by making sections of the program run in parallel. This can dramatically increase the performance of the code as the work is shared across multiple cores. With moder processors often having high core counts, this can also result in large performance gains. Similarly to vectorization, using OpenMP directives and environment variables parallelized code can be achieved with little changed to the implementation. In this running the critical sections in parallel resulted in a 13 times reduction in overall run time on the largest problem size.