Student number: 2101051

Optimizing an implementation of the D2Q9 algorithm using OpenMP

Joshua Gawley
kq21663@bristol.ac.uk

I. Introduction

In this report, we explore the optimization[1] and parallelization of a reference implementation of the *D2Q9 algorithm*. We show that whilst our serial optimizations and vectorization optimizations lead to better performance on all ballpark times, after parallelization, we achieve worse performance than ballpark times on small problem sizes but better performance than ballpark times on large problem sizes. We discuss the optimizations and why they lead to better performance, and we also analyze the scaling behaviour of the parallel implementation as we increase the number of cores.

II. Serial Optimizations

We first focus on applying serial optimizations. Out of the box, the reference implementation processed a 128x128 input in 28.97 seconds; we will use this as our reference point in applying optimizations.

A. Compiler choice and link-time optimization

The first (and most significant) serial optimization we made was to change from using the GNU compiler to the Intel compiler. The Intel compiler on BlueCrystal4 is more recent than the version of the GNU compiler on BlueCrystal4, which dates from 2021. Moreover, the Intel compiler supports more optimizations specifically for Intel processors. Therefore, we achieve a 37% speedup, with a runtime on the 128x128 problem size of 21.16 seconds.

The next serial optimization was to enable *link-time optimization* using the `-flto` flag. When link-time optimization is enabled, the compiler defers the optimization passes to the linking stage, allowing the entire program to be considered when applying optimizations. Hence, the compiler can apply optimizations such as function inlining more aggressively. After enabling link-time optimization, we achieved a 10% speedup, with a runtime on the 128x128 problem size of 19.11 seconds.

B. Constants and optimizing equilibrium density calculations

Next, we found that the value of many variables were not changed after being declared so we marked these variables as constants using the `const` keyword. This allows the compiler to apply constant propagation and constant folding to these variables during compilation, further reducing the amount of dead code and thus the number of instructions that the processor executes. Moreover, we also changed the equilibrium density calculation to use loops rather than inline assignments, allowing the compiler to apply vectorization to these calculations. These optimizations resulted in a 5% speedup, reducing the 128x128 runtime to 18.23 seconds.

C. Optimizing the timestep calculation

We now move on to refactor the `timestep` function more substantially. After inlining the `accelerate_flow`, `propagate`, `rebound`, and `collision` functions into the `timestep` function, we then saw that we could calculate the average velocity within the `timestep` function. Implementing this change saw a 20% speedup, as the 128x128 runtime dropped from 18.23 seconds to 15.32 seconds.

We then focused our attention on the relationship between `cells` and `tmp_cells`. In the original implementation, the contents of the `cells` structure were copied into the `tmp_cells` structure in the propagation step, and then when calculating `rebounds`, the values of the `tmp_cells` structure would be copied back into `cells` if we encountered an obstacle at a given location.

$$\texttt{cells} \xrightarrow{\text{copy}} \texttt{tmp\_cells} \xrightarrow{\text{copy}} \texttt{cells}$$

Fig. 1: Data movement per timestep in original implementation

This approach was necessary when the propagation and rebound calculations were separated into separate functions, but now that all steps happen in the `timestep` function, we can remove these unnecessary copies.

We do so by changing the calculations in `timestep` so that we always write from *source cells* to *destination cells*:

$$\texttt{src\_cells} \xrightarrow{\text{copy}} \texttt{dst\_cells}$$

Fig. 2: Data movement per timestep after refactor

If an obstacle is encountered, then both the propagation and rebound calculations are performed at once by directly assigning from `src_cells` to `dst_cells`. If an obstacle is *not* encountered, the speeds are copied into a temporary array, and then the collision calculations are performed on the temporary array before being assigned to `dst_cells`.

We then change the timestep loop so that on every other timestep, we copy from *destination cells* back into *source cells*. Whilst we could do this by swapping pointers, we actually change the timestep loop to do two timesteps per iteration (see Listing 1). This has the advantage of halving the number of iterations and thus the number of branches.

The removal of the extra copy in each timestep and halving of timestep iterations was very impactful; we saw a 33%

[1]We use Oxford spelling in this report.

speedup with our 128x128 runtime dropping to 11.5 seconds, below the given ballpark time.

```
for (int tt = 0;
     tt < params.max_iters;
     tt += 2)
{
        av_vels[tt] = timestep(params,
            src_cells, dst_cells, obstacles);
    av_vels[tt + 1] = timestep(params,
            dst_cells, src_cells, obstacles);
}
```

Listing 1: New timestep loop

D. Other optimizations

Other minor optimizations we made included changing the `obstacles` to store unsigned 8-bit integers, since these are small enough to store booleans but will facilitate vectorization.

After these optimizations, we see that across all problem sizes, we have achieved almost a 25% increase in performance compared to the ballpark times provided.

TABLE I: Runtimes after applying serial optimizations

Problem size	Ballpark time	Actual runtime	Speedup
128x128	14.3sec	11.5sec	1.24x
128x256	28.7sec	23.3sec	1.23x
256x256	114.9sec	92.9sec	1.24x
1024x1024	494.9sec	398.2sec	1.24x

III. Vectorization

Next, we focus on leveraging vectorization as much as possible. As we will see, our work

Running VTune on our implementation with serial optimizations, we observe the following analysis when it comes to vectorization.

```
Vectorization: 29.9% of Packed FP Operations
    Instruction Mix
        SP FLOPs: 46.9% of uOps
            Packed: 29.9% from SP FP
                128-bit: 29.9% from SP FP
                256-bit: 0.0% from SP FP
            Scalar: 70.1% from SP FP
```

Listing 2: Vectorization analysis after serial optimizations

We see that 70% of floating point operations are being done with scalar instructions, so there is more work to be done to increase the number of vectorized floating point operations.

A. More compiler flags

The first vectorization-related optimization we implemented was to use the `xHost` compiler flag which, among other things, asks the compiler to emit AVX2 instructions where possible. The advantage of using AVX2 instructions compared to SSE instructions is that AVX2 instructions operate on 256 bits at once instead of 128 bits, which means that in one instruction, the processor can operate on eight single precision floating point at once. This additional compiler flag led to a 50% speedup, with the runtime of the 128x128 case decreasing to 8.69 seconds.

B. Forcing auto-vectorization and using array of structures

We then attempted to enable more auto-vectorization by using the `omp simd` directive on key loops such as the timestep calculation. Whilst this lead to 99% of floating point operations being vectorized, also it caused a huge regression in performance; the runtime for the 128x128 problem size after forced auto-vectorization was 19.46 seconds.

We found that the cause of this regression in performance was due to the memory layout of the `t_speed` structure; the original implementation used a *array of structures* memory layout as seen in Listing 3. This means that we have 2D grid of pointers, each pointing to an array of speeds.

```
/* struct to hold the 'speed' values */
typedef struct {
  float speeds[NSPEEDS];
} t_speed;
```

Listing 3: Original definition of `t_speed`

The problem with this is that when we assign new speeds in the timestep calculation, we end up with strided memory access (in this case, a stride of length 9). This is highly unperformant when vectorized because the processor operates on four cells at once but has to make four memory accesses to get the speed values because, say, each `speed0` is in a different cache line.

To solve this, we change to using a *structure of arrays* layout, where we store arrays of speeds which then are indexed by cells. Then, the `speed0` of each cell will be stored contiguously in memory, so only one memory access is required when loading, say, four of these for the timestep calculations.

```
/* struct to hold the 'speed' values */
typedef struct {
  /* each array is of size
     params.ny * params.nx
  */
  float *speed0;
  float *speed1;
  float *speed2;
  float *speed3;
  float *speed4;
  float *speed5;
  float *speed6;
  float *speed7;
  float *speed8;
} t_speed;
```

Listing 4: New definition of `t_speed`

An additional change we made at this stage was to manually align the `src_cells` and `dst_cells` arrays to 32 bits; this ensures that every eight speed values will be on the same cache line (since a `float` variable in C is 32 bits large on the `x86-64` architecture). We see that after these changes, the compiler vectorizes pretty much all floating point operations:

These changes reversed the performance regression and actually resulted in a 38% speedup compared to when we

enabled AVX2 optimizations, reducing the runtime to 6.23 seconds.

C. Eliminating branching

The final change we made was to switch from using branches to masking. In the original code, the code branches based on if an obstacle is encountered or not. However, branches interfere with vectorization, so we refactored the timestep code to eliminate branches by the use of masking. This lead to a 6% speedup, reducing the runtime of the 128x128 case to 5.95 seconds.

After all of these changes we see that the compiler now successfully autovectorizes almost all floating point operations.

```
Vectorization: 99.9% of Packed FP Operations
    Instruction Mix
        SP FLOPs: 21.3% of uOps
            Packed: 99.9% from SP FP
                128-bit: 5.3% from SP FP
                256-bit: 94.6% from SP FP
            Scalar: 0.1% from SP FP
```

Listing 5: Vectorization analysis after faciliating vectorization

Moreover, we see that we achieve roughly a 20% increase on the ballpark times after vectorization.

TABLE II: Runtimes after applying serial optimizations

Problem size	Ballpark time	Actual runtime	Speedup
128x128	7.1sec	6.0.sec	1.18x
128x256	14.4sec	11.9sec	1.21x
256x256	57.7sec	47.3sec	1.22x
1024x1024	287.5sec	237.2sec	1.21x

IV. Parallelization

All of the optimizations we have implemented up until now has been focused on maximizing single-core performance. However, a single node on BlueCrystal4 gives us 28 cores to play with, so we now focus on parallelizing our code.

A. OpenMP directives and leveraging first touch policy

Our first parallelization involved adding `omp parallel for` directives to the outer timestep calculation loop, initialization loops, and loop calculating the total density. We retain the `omp simd` directives on inner loops to retain vectorization.

Parallelizing the initialization loops may seem pointless at first glance, but as discussed in labs, this exploits the first touch policy of the OS memory allocator, ensuring that each individual thread touches the same data in both the initialization and timestep loops. Moreover, we use the `reduce` directive to enable parallel reduction of the `tot_u` and `tot_cells` variables in the timestep calculation in order to ensure correction. After this first parallelization effort, we achieved a 500% speedup on the final autovectorized code, with the 128x128 case running in 1.2 seconds.

B. Thread pinning

The parallelization of the initialization loop makes our code NUMA-aware but by default, the OS can move threads between cores. In order to prevent this and thus minimize remote memory accesses, we set the `OMP_PROC_BIND` to `true` and `OMP_PLACES` to `core`. As discussed in labs, setting `OMP_PROC_BIND` ensures that threads are not moved, and setting `OMP_PLACES` to `cores` sets the affinity policy so that each thread is run on a single hardware core. With this optimization, we achieved a 5% speedup, with a runtime on the 128x128 problem size of 1.14 seconds.

C. Optimizing the distribution of loop iterations among threads

The final optimization to the parallel case was to specify the distribution of loop iterations among threads using the `schedule` directive; we added the `schedule(static)` directive which divides the iterations into chunks and assigns these chunks in a round-robin manner, as discussed in labs. This is usually the fastest option, and indeed after implementing this, we achieved a 10% speedup, with a runtime on the 128x128 problem size of 1.03 seconds.

Surprisingly, we tried other optimizations, such as using the `restrict` keyword to guarantee that pointer aliasing does not occur, using `__assume_aligned` directives to tell the compiler that the arrays are indeed 32-bit aligned, and using the `collapse(2)` directive to convert two loops iterating over cells to one loop, but we found that actually decreased performance. Therefore, we have the following runtimes for each problem size after parallelization:

TABLE III: Runtimes after applying OpenMP parallelization

Problem size	Ballpark time	Actual runtime	Speedup
128x128	0.7sec	1.0sec	0.70x
128x256	0.9sec	1.19sec	0.86x
256x256	3.1sec	3.5sec	0.89x
1024x1024	14.7sec	11.8sec	1.25x

Bizarrely, we find that on the smaller problem sizes, we have not quite met the ballpark times, but on the largest problem size, we have beaten the ballpark time with a similar speedup to when we implemented serial optimizations and vectorization. We will discuss the

V. Performance analysis

We now analyze the performance of our optimization code. We look at various aspects such as parallel scaling and computational intensity, using a Roofline model to compare the performance characters of each phase of optimization.

A. Parallel scaling

We ran a hotspot analysis on the original implementation (see Listing 6) and we see that the percentage of code we ended up parallelizing was 97.6% (i.e. by far the most time was spend calculating each timestep).

Therefore, up to around 32 threads, we would expect as we increase the number of threads, the speedup compared to the serial case increases in a roughly linear fashion (above 32 threads, the speedup will increase in a more logarithmic fashion).

Top Hotspots

Function	...	CPU Time	% of CPU Time(%)
collision	...	19.410s	69.7%
av_velocity	...	4.250s	15.3%
propagate	...	3.470s	12.5%
main	...	0.670s	2.4%
_IO_fprintf	...	0.030s	0.1%
accelerate_flow	...	0.020s	0.1%

Listing 6: Hotspot analysis of original code

Now we show the parallel scaling behaviour we actually observed in Fig. 3.

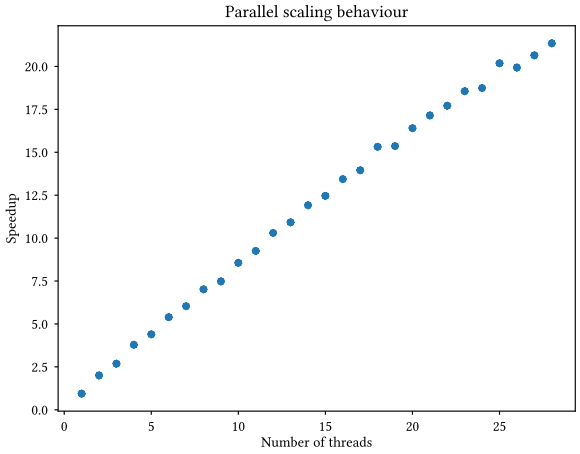

Fig. 3: Speedup vs number of threads (using 1024x1024 problem size)

As we expected, we see that the speedup increases roughly linearly as we increase the number of threads. Further investigation on larger compute nodes would be needed to discern where the speedup increase starts to tail off.

B. *Roofline analysis*

We now look at a Roofline plot (Fig. 4) showing the performance of the timestep loop after each phase of optimization (we used Intel Advisor to find these values). We ran all tests on the 128x128 problem size, except for an additional test of the parallel implementation on the 1024x1024 problem size to decipher possible causes of the idiosyncratic performance profile we saw in Table III (NB: the "roof" in this case is peak performance on single precision floating point numbers).

We see that the computational intensity remains roughly constant whilst the performance increases after each phase of optimization. Interestingly, we see that in the parallel implementation, the 128x128 problem size was memory bandwidth bound whereas the 1024x1024 problem size was cache bound. A possible reason for this is that the 128x128 problem size is so small that when we parallelize to 28 cores, the computation becomes dominated by memory synchronization and context switches between threads , which may not be such a problem when dealing with the 1024x1024 problem size as this is 64 times larger than the 128x128 problem size.

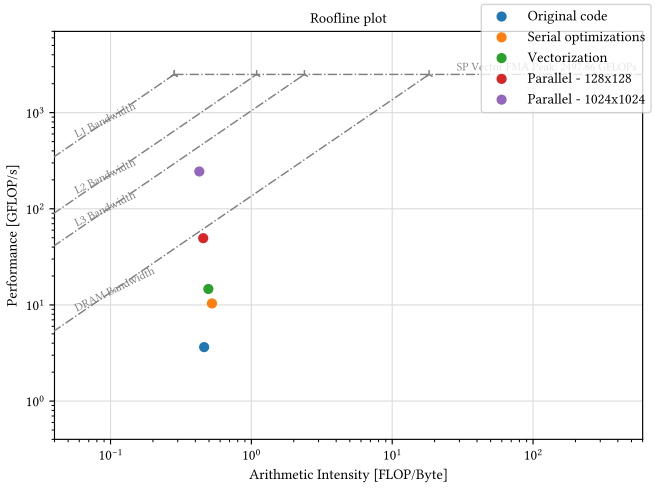

Fig. 4: Roofline plot

C. *Future work*

We now look at how further optimizations could be made. Based on the Roofline analysis, increasing raw performance may not be possible as the parallel implementation is already cache bound, and optimizing cache efficiency may not be possible because at least three instructions are needed to load all nine speeds of a cell (two instructions are needed to load eight single precision floating point numbers if vectorization is used). Therefore, future efforts should be focused on improving the computational intensity of the implementation

VI. Conclusion

We implemented a number of optimizations to the D2Q9 algorithm, moving from serial optimizations to vectorization and finally parallelization using OpenMP. We achieved better runtimes than the ballpark times after serial optimization and vectorization, whereas our parallel implementation was slower than ballpark times for smaller problem sizes and faster than ballpark times for larger problem sizes. We analyzed the performance of our implementation, comparing the parallel scaling behaviour against what would be predicted theoretically and then using a Roofline analysis to understand the performance characteristics of each phase of optimization, especially the peculiar performance profile of the parallel implementation. We concluded with speculating on possible strategies for finding future optimizations based on the Roofline analysis.