Advanced High Performance Computing: Optimising Lattice-Boltzmann with OpenCL

Michael Rollins (mr16338)

1 Introduction

This report will document the optimisation techniques performed in order to port provided Lattice-Boltzmann code for fluid simulation to OpenCL to be run on a NVIDIA K20m GPU on BCp3. This involved first moving each of the loops in the functions within the `timestep()` function to kernels within a kernels.cl file to be ran on the GPU cores, then optimising these functions, along with various serial optimisation techniques in order to reduce the run time of the code as much as possible. I first selected the compiler and optimisation flags I would use, from a small amount of experimentation I found that using the ICC 16.0.2 compiler with flags -Ofast and -xAVX, which respectively maximise the amount of optimisation performed by the compiler and generate SIMD instructions based on the architecture of the CPUs within BCp3. During this report I used work groups of size 32 rows and 4 columns as I found these sizes to be optimal, which is discussed towards the end of the report.

2 Optimisations

2.1 Porting Function Loops to Kernels

The skeleton code we were provide with had the loops within the `accelerate_flow()` and `propagate()` functions already moved into the kernels.cl file. This left me to port the `rebound()`, `collision()` and `av_velocity()` functions. Kernel functions are run on each point within a problem domain, in the case of the Lattice-Boltzmann code the kernels run on each cell within the cells grid. GPUs are throughput optimised, as opposed to CPUs which are optimised for latency, they do this by having far more cores than CPUs, running many operations at once across many cores. In the case of the NVIDIA K20m GPUs they have 2496 processing elements over 13 cores. This allows for far greater parallelism than on can achieve with CPUs.

Moving the `rebound()`, `collision()` loops to kernels was simple as it involved repeating the same steps which were demonstrated in the skeleton code. Moving the `av_velocity()` function was slightly more complicated as it required performing a simple reduction of the `tot_u` variable which accumulates the magnitudes of the velocities of the cells. To do this, within the kernel I used a `barrier(CLK_LOCAL_MEM_FENCE)` call to make all work items within a work group wait until their calculations were finished. I then used one work item from the group to sum all of their velocity magnitudes within their work group, which are stored in an array local to the work group. Each work group then places the sum of all their velocity magnitudes into an array in the global space. Within the c `av_velocity()` function, these are then totalled into the `tot_u` variable. This reduction was adapted from a 1D reduction found in the solution to exercise 9 of the HandsOnOpenCL course Github [1]. I had originally done the same thing for the `tot_cells` variable (which holds the total number of non-obstacle cells) as well, however I discovered that the amount of calculation within the timesteps could be reduced by calculating this once during the `initialisation()` function, as this does not change during the program.

Having implemented these changes the runtimes of the program on grids of size 128x128, 256x256, 1024x1024 and 128x256 were, respectively, 25.9s, 160.7s, 559.3s and 50.2s.

2.1.1 Removing Unnecessary Reads and Writes

Having done this I realised as no computation was being done during the `timestep()` function within the c program, that there was no need to read or write `cells` or `tmp_tells` to and from the global memory. Previously around each `timestep()` call the cells would be written to the GPU and then read back to the CPU. Instead I have only a single write before the main computation began, to send the original grid configuration to the global GPU memory, then a single read, after the computation was complete to transfer the grid back into CPU memory.

As transferring data from CPU memory to the global GPU memory, and vice versa, has a high latency, removing these unnecessary functions made a significant improvement to run times. On all grid sizes the speed up was of around 3 times, with slightly larger speed ups on the smaller grids. The new runtimes on grids of size 128x128, 256x256, 1024x1024 and 128x256 were 7.02s, 53.3s, 173.6s, 14.39s.

2.2 Merging Kernels

The next step I took to speed up the code was to merge the `propogate(), rebound(), collision() and av_velocity()` kernels, into one kernel. This is allows for the majority of the computation to be done in one kernel on the GPU, rather than having to come back to the CPU code, setting up a new kernel's arguments and adding it to the queue. As well as reducing the overheads of setting up and running kernels, by combining these kernels I also reduced the number of times the program needed to iterate over the grid, as all four kernels iterate over the entire grid, where as with them combined, only one iteration was needed to complete all of their computation. By only iterating over the grid a single time the number of memory operations was also reduce, and as the code is memory bandwidth bound this resulted in further improvement in the runtimes. Once again for the grids of size 128x128, 256x256, 1024x1024 and 128x256 then runtimes were 4.04s, 53.1s, 121.56s, and 15.00s.

2.3 Pointer Swaps

In the original code, like within the previous assignment, the grid was copied from `cells` to `temp_cells` and back again with every iteration. For each iteration of the `timestep()` function one of the these copies was unnecessary, so by removing this I could once again reduce the number of memory operations necessary. In doing this however it left the current working grid in the `tmp_cells` variable, which would not normally be worked upon on the next iteration (using the standard code it would always start operating on `cells`). Therefore in order to have every second iteration act on the `tmp_cells` variable first, I made copies of both the main computation function and the `accelerate_flow()` function which reverse the `cells` and `tmp_cells` kernel arguments and had these functions be ran instead of the originals on every second iteration. On the smallest grid there was only a small improvement of around 0.5s or approximately 12.5%, however on the larger grids there was a far greater improvement; on the grid of size 256x256 there was nearly a 4 times speed up to 13.78s, and on the 1024x1024 grid the speed up was just over 4 times to 29.29s. The grid of size 128x256 also had a large improvement in run time at this point, dropping to 5.43s which is just over a third of its previous run time. **LOOK INTO WHY**

2.4 Changing Data Structure

The original code that was provided had the grid stored in an Array of Structures (AoS) data structure, this means the grid was an array which contained a struct, holding each of the 9 speeds, for every cell. This is a non-optimal solution as it does not allow as much coalesced memory access as a Structure of Arrays (SoA) layout. **WHY NOT?** By changing the data structure into a SoA style array, where all the speed0 elements are stored together, followed by all the speed1 elements and so on, all speeds of the same direction are stored contiguously in memory. This change allows for coalesced memory access as when a work item needs to load a speed from memory, it will load a chunk of memory which will contain the speeds needed by the other work items in a work group, as they are stored contiguously. This sharply reduces the number memory accesses needed compared to if the speeds were interleaved.The speeds of the 128x128, 256x256, 1024x1024 and 128x256 sized grids at this point were 2.27s, 8.92s, 16.81s and 3.13s.

2.5 Improved Reduction

I realised that the reduction I was using to sum the `tot_u` variable was not as fast as it could be. My original reduction had one work item in a work group go through every cell that the work group was operating on and reduce it into a single variable. This is not optimal as it is possible to spread the work across multiple work items. The new reduction works by having the first half of the work items, adding there counterpart in the second half of the work group to themselves, by means of having a `stride` variable equal to half the work group size, so each element with index `x` would perform `cell[x] += cell[x+stride]`, thus halving the size of work items the totals are stored in. Then by halving the `stride` variable, the same operation can be repeated until the full total of all the velocity magnitudes is stored in a single store which can then be read back to the CPU memory to be totalled with the magnitudes of the other work groups. So rather than the original function which took approximately O(x) time, where x is the work group size, the improved reduction only takes $O(\log_2 x)$time. The reduction I used in my

code was a adapted from a 1 dimensional reduction written for OpenCL found on https://dournac.org [2]. This improvement reduced the runtimes to 1.95s, 7.21s, 14.7s, and 2.93s.

2.6 Masking

Conditional statements within kernels written to be run by GPUs have a high latency, this is because GPUs do not usually support speculative execution which means all branches must be executed, and if an individual work item does not participate in that path it simply ignores it, but has to wait until the other work items have completed the branch. In order to reduce this latency we can convert conditional code into straight line code by using masking. This involves taking the conditional statement and using it as an arithmetic value to alter the results of an expression based on whether it returned true or false. This would change a piece of code in the style of

```
if(x == true) {y += 1} else {y = 0}
```

into this:

```
int mask = (x == true); y = (mask) * (y + 1);
```

By replacing the large number of conditionals in my kernals with this style of code I could vastly reduce the number of branches in the execution path reducing the amount of time that some work items are not performing calculation for. Having finished this optimisation I achieved the fastest time that I was able to produce. For the grid of size 128x128 my fast time was 1.76s, for 256x256 it was 5.97s, for 1024x1024 it was 13.03s and for 128x256 it was 2.25s

3 Analysis

3.1 Finding Optimal Work-Group Size

At the start of this report I wrote that that I was using work groups of size 32 by 4, I decided on this by running the code multiple times using different work group sizes and comparing the results. I found that when number of rows was greater than the number of columns the result was always superior to if the values were reversed. This is due to the memory access of the work groups as the rows are stored contiguously in memory so the they allow for coalesced memory access. It also became clear that when the row size was multiple of 32 the results were faster than otherwise. Below I have compiled a table of the best work group sizes I found when running the code on the 128x128 grid.

Rows	Columns	Runtime
32	1	1.933992
64	1	1.819162
128	1	1.758264
32	2	1.837189
64	2	1.766025
128	2	1.816
32	4	1.755701
64	4	1.792623
128	4	1.898686
32	8	1.818124
64	8	1.885497

3.2 Comparison to Other Lattice-Boltzmann Implementations

Having in the previous assignment already produced Lattice-Boltzmann implementations using both serial code and utilising the OpenMP interface, it is interesting to compare the runtimes across these different methods.The best times I achieved with optimised serial code for grid sizes 128x128, 256x256 and 1024x1024 were 7.67s, 62.43s and 272.39s. When I used OpenMP to utilise several processor cores to run the code in parallel on CPUs, I managed to reduce these run times to 0.93s, 5.50s and 33.44s. It then becomes obvious that using OpenCL to run the code on GPUs is not always the most optimal solution for the Lattice-Boltzmann code, on the grid of size 128x128 it is almost twice as slow as using optimised OpenMP code. On the 256x256 grid the times are within 10% of each other but the OpenMP code was still faster. It is only when running on the large grid of size 1024x1024 that we begin to see why OpenCL and GPU programming in general can be so powerful. Due to the vast number of processing

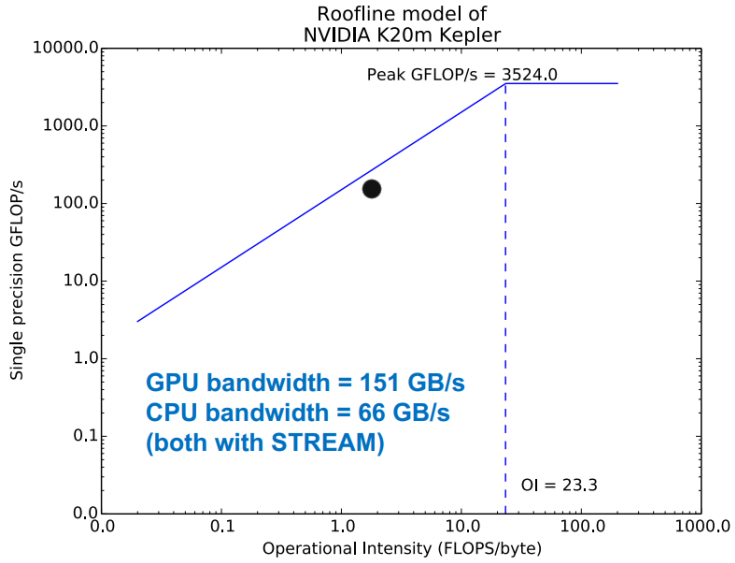
elements in a GPU, they are optimised for programs being run on a large number of items, as they are optimised for bandwidth over latency. They can spread the processing of the items across a large number of work groups and with in them work items, allowing for much larger amounts of computation to be performed simultaneously than is possible with a conventional CPU. However when on a smaller grid the benefits of having such a large number of processing elements is outweighed by the the advantage CPU's have of such low latency, meaning on smaller grid sizes OpenMP is a more effective way of implementing Lattice-Boltzmann, but on increasing large grid sizes, using OpenCL on a GPU becomes the faster way of running the code.

3.3 Analysis Using the Roofline Model

The maximum achievable bandwidth of a system can be measured using the STREAM [3] benchmark. By using the result of this benchmark of the NVIDIA K20m Kepler in BCp3, we can compare the results I managed to achieve to what would be theoretically possible if the entire potential bandwidth was used. The peak bandwidth for these GPUs is 151GB/s. We can calculate the bandwidth achieved by the Lattice-Boltzmann code by first calculating the total memory usage for the code, and then dividing it by the run time the code produced. The memory usage can be calculated as such:

$$\frac{sizeof(float) * size\_of\_grid * NSPEEDS * 2 * maxIters}{runtime}$$

If I insert the values for largest grid size, which utilised the GPUs hardware the most the result is `4 * 1024 * 1024 * 2 * 20000 / 13.03 = 115.9GB/s` which is approximately 76.7% of the maximum achievable bandwidth. To correctly measure the performance of the code in GFLOPS/s it is necessary to multiply the bandwidth by the operational intensity of the code. This is calculated by dividing the number of floating point operations needed per iteration by the number of memory operations. I calculated the number of floating point operations to be 122, and the number of memory operations is 9 loads and 9 writes (for 9 speeds are loaded from `cells` and written to `tmp_cells`). Therefore the operational intensity was `122/(18*4) = 1.69`. Therefore by multiply the bandwidth by the operational intensity we can calculate the performance to be 195.9GFLOPS/s. Below I have shown this drawn onto the roofline model of the NVIDIA K20m GPU.



References

[1] Simon McIntosh-Smith and Tom Deakin. HandsOnOpenCL, https://github.com/HandsOnOpenCL/Exercises-Solutions.

[2] Fabien Dournac, https://dournac.org/info/gpu_sum_reduction

[3] Department of Computer Science, University of Virginia. http://www.cs.virginia.edu/stream/ref.html