Advanced High-Performance Computing

Lattice Boltzmann OpenCL Implementation

Tyler Ward (tw17530)

Introduction

This report details the process I went through to implement the Lattice Boltzmann code onto OpenCL. The original code I was given already had two of the five main functions ported to OpenCL. I began by porting the rest of these over to OpenCL before progressing with further optimisations. These optimisations included fusing the different functions into one, pointer swaps, changing the layout array of structures to structure of arrays and reducing the unnecessary read and writes to the kernel. I will go into more detail about these optimisations perform an analysis of the performance of the code in comparison to the OpenMP version.

Porting functions to OpenCL

In the original code supplied, the functions `accelerate_flow()` and `propogate()` were already ported to OpenCL. The next two functions I ported to OpenCL were `rebound()` and `collision()`, which were done in a similar way to the first two provided functions. In the main program file, I first added the kernels to the t_ocl struct and then created the kernels in the `initialise()` function, as well as making sure they were released after the program was finished processing. After the kernels were initialised and, I had to change the functions to remove the body of them and replace it with code to set the kernel arguments and enqueue the kernel. Initially I also added the `cl_finish()` function, but as I will explain later in the report, this was not needed in every function. All that was left to do was put body of the function into the kernels file and add in the functions to find the variables ii and jj instead of the `for` loop. These two functions were pretty straightforward to port to OpenCL.

The fifth and final function left to port to OpenCL was the `av_velocity()` function. This function required a few extra steps in order for it to produce the correct output as different work groups cannot be synchronised and each work item has its own memory. Since the `av_velocity()` function has to total all of the velocities and cells, I had to go about calculating the average in a different way to what is originally in the code. First of all, I had to add a reduce function. The reduce function totals up all of the velocities and cells from each work item in a work group using a memory fence to allow for the synchronisation of each work items' memory. For the function to work, I had to add in four more arrays to hold both the local memory totals in each work group and then the global work group totals. The reduce function calls on one work item to sum up the local work group totals and writes it to the global array. The global array is then read in the main program and totalled up to give the final average velocity value.

Loop Fusion

Loop fusion is one of the optimisations also made in the OpenMP optimisation of this code. The idea of loop fusion is to put all of the function bodies into one loop in order to reduce the amount of times that the program has to iterate through the cell array. The process of doing this was pretty similar to how it was done for the OpenMP implementation in terms of the kernels file. For the main file, there were some differences compared to the kernel enqueueing in the first 4 functions ported. These functions only required a global size for enqueueing the kernel, however, with the introduction of the `av_velocity()` function to the loop, I had to provide a local size as well. I acquired the size I had to set this to by using the `clGetKernelWorkGroupInfo()` function and square rooting the returned value to give the size of each side of the work group as it needed to be given as 2 dimensional.

Removing Unnecessary Kernel Calls

In the original code given, the first two functions that were ported to OpenCL didn't have a very efficient kernel calling layout. By this, I mean that the cells were written to the kernel and read from the kernel once each time step. Reading and writing the cells to the kernel is a very expensive operation to perform. So once, all of the functions were ported to OpenCL, I only needed to write the cells to the kernel once at the beginning and read the cells from the kernel at the end of the last time step. I had to wait until all the functions were ported over so that I did not need any cell information in the main program in between each time step.

The same idea applies for the global total velocity and cells arrays. Initially I had these arrays being read from the kernel after each timestep in order to calculate the average value to put into the `av_vels` array. It isn't necessary to update the value each time step, so instead I took the reads out of the main process function and put them, along with the totalling code, after the time step loop. However, I couldn't just move the kernel reads for the program to still work properly. I had to extend the size of the global arrays for it to store the local memory totals for each time step. Along with this I had to change the indexing in the reduce function to put the values into the correct time step.

As previously mentioned earlier in the report, the `cl_finish()` function isn't required at the end of every function or time step. Similarly to the kernel reads and writes, `cl_finish()` is quite an expensive function if you're calling it at the end of each function every iteration. OpenCL only requires you to call `cl_finish()` once at the end of kernel processing, so I removed it from the `av_velocity()` and `process()` functions and moved it to after the cells and average velocity information had been read.

Layout Change to Structure of Arrays

Arguably the most effective optimisation was changing the layout of the code from array of structures to structure of arrays. This involved changing the way the cells are held from using the t_speed struct to using a float pointer. The first thing I had to change was the way the grid is initialised. In the `initialise()` function, I had to alter the way that the cells and the speeds for each cell are indexed. Instead of the index in the array pointing to a struct holding each of the 9 speeds, I changed the indexing in the array to instead point to a specific speed of a certain cell.

After changing the way that the cells are initialised, I then had to carefully go through the rest of the code to ensure that the new indexing system was consistent over the rest of the program. This process can be a cause of a lot of errors as the cell indexing is the most important part of the program and accessing one wrong cell can cause the whole program to output incorrectly. This required extreme care and constant checking that the correct speeds and cells were being accessed everywhere.

The idea behind switching the layout to structure of arrays is to encourage coalesced access. When something is pulled from memory, it is given to the program in a cache line. The initial array of structures layout holds the speeds for each cell such that when each speed is called it is taking in a cache line of information it doesn't need along with it. The structure of arrays layout allows for all the speeds to be given to the program once the first speed is accessed from memory, as they are given to the program in the cache line. This mean that not only is there less unused information being passed which is an expensive operation, but there will be fewer calls to memory as the program doesn't have to access memory for every single speed call as it will already have the information available in local memory.

Pointer Swaps

Finally, the last optimisation I implemented was in the kernel file. At the beginning of the main function that calculates all of the cell updates, cells are loaded into tmp_cells and back again later in the function, as well as tmp_cells being accessed to be used in calculations. It becomes very expensive to access memory to get the values in cells and tmp_cells if we are doing it so frequently. To stop this from happening, I initially loaded the values into local variables so that they are available in local memory for calculations. This reduces the calls to memory to access cells or tmp_cells, however, it now brings in the issue of a race condition. When loading the values into tmp_cells there are rules the program follows to stop a race condition occurring. After implementing the local memory optimisation, the program had errors due to the fact that some values were being written back to cells before the previous values were read. This called for me to change the function to put the final state into tmp_cells and then perform a pointer swap in the main program before the next time step is called. This prevented a race from happening as neither grid was both being accessed from and written to during the function.

Analysis

The final runtimes I achieved for each of the grid sizes 128x128, 256x256, 1024x1024 are respectively as follows: 0.86s, 2.14s, 4.46s. In comparison to the OpenMP implementation of this code, the biggest differences can be seen in the larger grid sizes.

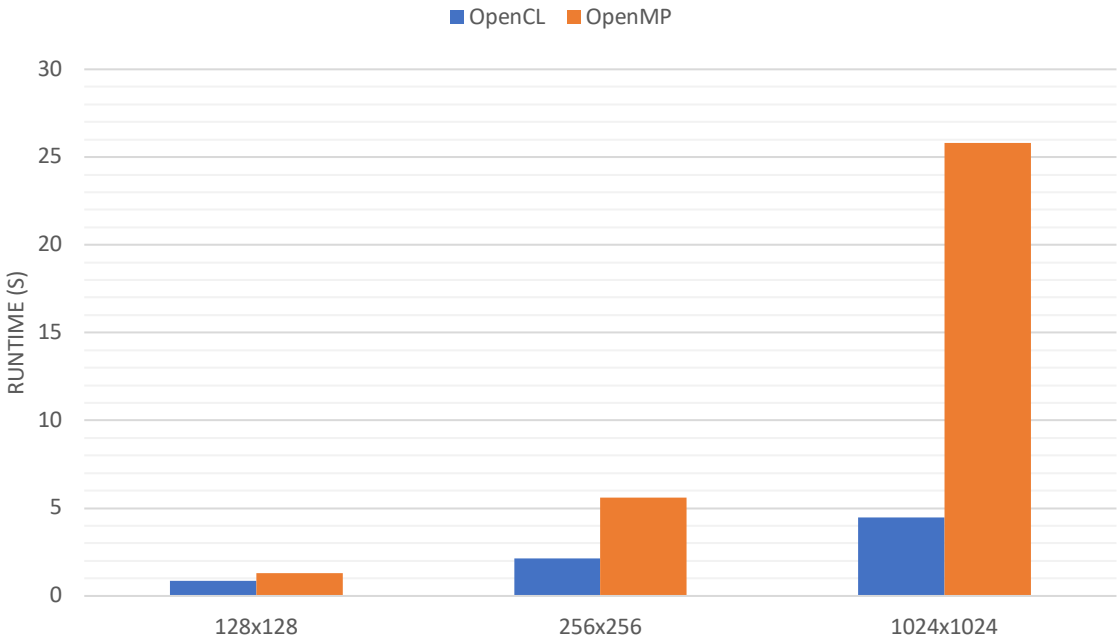

Figure 1.

Figure 1 above shows the runtimes for each grid size on OpenCL compared to OpenMP running on 28 cores. As you can see, the difference between OpenMP and OpenCL on the 128x128 grid is very minimal. The speed of the OpenCL implementation is really emphasised when you look at the larger grid sizes. The OpenCL implementation has a 5.8x speedup compared to the OpenMP implementation with the runtime on the 1024x1024 grid on OpenCL even being faster the OpenMP runtime on the 256x256 grid. This really shows how much better the GPU handles large amounts of operations compared to the CPU, even with vectorisation and other optimisations including aggressive compiler optimisations.