HPC Coursework 3: **Using OpenCL to Optimize The Lattice Boltzmann Program**

1.0 Introduction

This report looks into the optimizations implemented on a program that implements the **Lattice Boltzmann** scheme using the *OpenCL* libraries. The times mentioned in the report pertain to testing on the 1024x1024 parameters and obstacles files unless and otherwise mentioned. Before looking at any **Compiler Optimizations** that are possible, we will implement a naïve version of the program on *OpenCL*, which minimally takes advantage of any of the advanced parallelizing features that *OpenCL* possesses. Then we will proceed to look at any **Serial Optimizations** that can be made to the program, before looking at how we can achieve the best speed using *OpenCL* in the section **Parallel Optimizations**.

2.0 Naïve Implementation with OpenCL

2.1 Kernel Design

Each of the previous functions (*accelerate_flow, propagate*, *rebound*, *collision* and *av_velocity*) were converted into kernels. Since *accelerate_flow* functions on only one row, it would be more logical to use a 1D kernel. No changes were made to the serial code on most of the kernels except for the replacement of *for* loops pertaining to the *ii* and *jj* (where appropriate) counters with *global_id*'s *ii* and *jj*. Collision and av_velocity were modified to contain a naïve implementation of reduction (in this assignment *av_velocity* has been combined with collision for convenience and optimization). While local sums of work groups are calculated within *collision*, the global sums are reduced to one answer in *av_velocity*. It must be noted that the local size is being defined as {***32***} when pertaining to one dimension and {***32, 1***} (32 columns within one row in a workgroup) when dealing with two dimensions.

2.2 Memory Management and Reduction Technique

Four buffers for *cells*, *tmp_cells*, *obstacles* and *av_vels* were created to be set as kernel arguments. The assignment of *cells* and *tmp_cells* as arguments to the kernels alternate every iteration. This way we avoid any possible data races involved with combining the kernels (described in **Section 5.1**). Two buffers namely *global_sums* and *d_iteration_cntr* were used for the reduction. *d_iteration_cntr* is a necessary one element array that remembers the iteration number to store the values added from *global_sums* into *av_vels*. *global_sums* is the array in which the sums of *u_sq* (*local_sums*)from each workgroup is stored. The reduction currently being used is a simple algorithm with an accumulator.

2.3 Profile of the Code

The total time the code takes with a naïve implantation is **291.78s**. After being manually profiled with timing code set between the different kernels for one iteration we get these results:

Table 1

Kernel	*accelerate_flow*	*propogate*	*rebound*	*collision*	*av_velocity*
Time taken (s)	0.000036	0.000193	0.000015	0.000020	0.000018
Percentage	**12.8%**	**68.4%**	**5.5%**	**7.1%**	**6.4%**

It can be seen that propagate takes the most amount of time and needs to be heavily optimized.

3.0 Serial Optimizations 1

4.1 Improving on *propagate*

Minding the fact that scaterred reads take a lot longer than scaterred writes, a closer look at *propogate* revealed that it worked in a pretty inefficient way. If we look at where the values of speeds from each cell goes in terms of the position of the cells of *tmp_cells*(blue arrows in **Figure 2** and 'speed' 0 stays in the same place), it is clear that the current model for reading into *tmp_cells* is pretty scaterred. Where we to turn this into a structure where each speed within a cell of *tmp_cells* is systematically read into, we see an improvement of **2s** to **289.03s**. The red values outside the box show the exact speed value (in terms of the position of the cells of *cells*) that's being transferred to be written into *tmp_cells*.

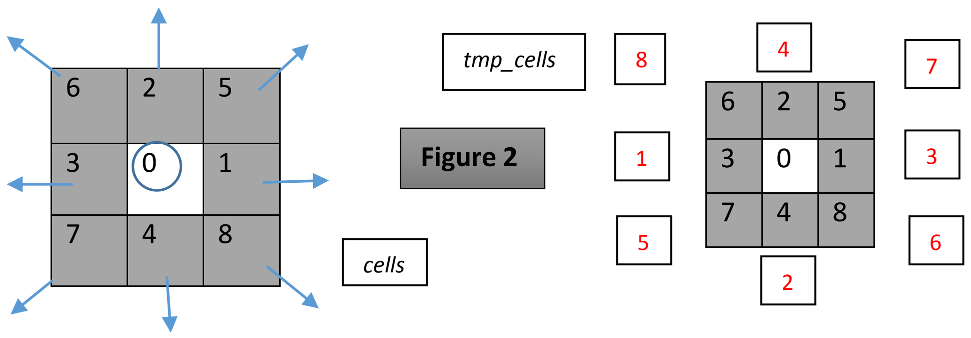

Figure 2

4.2 Improving on *collision*

collision can be optimized by calculating the inverse of *c_sq* before the iteration and calculate the inverse of *local_density* before we calculate the velocity components (we wouldn't have to divide the by local density twice). The original and the new equations have been shown below (**Equation 3**):

$$d_{equ[n]} = \mathrm{w1} \times \mathrm{local}_{density} \times \left(1.0 + \left(\frac{\mathrm{u[n]}}{c_{sq}}\right) + \left(\frac{\mathrm{u[n]} * \mathrm{u[n]}}{2.0 * \mathrm{c\_sq} * \mathrm{c\_sq}}\right) - \left(\frac{\mathrm{u\_sq}}{2.0 * \mathrm{c\_sq}}\right)\right)$$

$$d_{equ[n]} = \mathrm{w1} \times \mathrm{local}_{density} \times \left(1.0 + (\mathrm{u[n]} * \mathrm{inv\_c\_sq}) + \left(\frac{\mathrm{u[n]} * \mathrm{u[n]} * \mathrm{inv\_c\_sq} * \mathrm{inv\_c\_sq}}{2.0}\right) - \left(\frac{\mathrm{u\_sq} * \mathrm{inv\_c\_sq}}{2.0}\right)\right)$$

Equation 3

This improved my time to about **265.4s**.

4.0 Compiler Optimizations and Changing Data Type

We are currently using the gcc compiler with the *–ffast* and *–O3* flags. However when we remove the flags we see no improvement in the timing of the code whatsoever. This is mostly due the fact that these flags only work on the host code. Were we to investigate the inclusion of options during the *clBuildProgram()* stage, we get a change in the run time. After using the all combinations of flags such as -cl-mad-enable, it became very clear that there would be no optimization to the time. However after changing most of the *doubles* in the entire program and kernels (including the data structures) to *floats*, some of the flags became feasible. A combination of "*-cl-single-precision-constant -cl-no-signed-zeros -cl-mad-enable*" as the optimization flags and using floats instead of doubles improves the time to **223.4s**.

5.0 Tackling the 700x500 Grid

A simple if statement that tests whether or not the grid is a 700x500 grid and changes the local workgroup size to {**10**, **10**} or {**10**} (1 dimensional kernel) works for this sepcial grid.

5.0 Parallel Optimizations

5.1 Combining the Kernels

Since we ultimately want to reduce the data transfer between the memory and the host, combining the kernels would be the best option here. We have already made sure that no transfer happens between the start and the end of the iterations. Since the functions of *collision*, *propagate* and *rebound* are quite similar they can be combined into on kernel with a little bit of tweaking. This radically improved the run time to **175.4s**.

5.2 Using Private Memory

There has already been an instance of using local memory within this program for the local sums. It would be very beneficial for us to use private memory that stores the values of each cell that is being worked on for each work item. This is copied from *cells* (or *tmp_cells* depending on the iteration) and after it has been processed by the merged *collision*, *propagate* and *rebound* is copied back into *tmp_cells* (or *cells* depending on the iteration). This further decreased the time of the *propagate* kernel and the entire program to **132.6s**.

5.3 Coalescing the Memory

Since we are copying the contents of *cells* buffer into the private memory of each work item, it would be wise to make sure that accesses to the memory are coalesced. This required changing the way the data structures (*cells* and *tmp_cells*) stored information. The data type of t_speeds was removed altogether and the individual cells are laid out as shown below:

params.nx*params.ny speed 0	params.nx*params.ny speed 1	param.nx*params.ny speed 2	…

Figure 4

This led to a drastic improvement to **78.6s**.

5.6 Profiling 2

If we profile now we see that the main combined kernel is taking up most of the time (**Figure 5**). After trying to improve on the coalesced memory access by creating a local work group that copies the values needed for the work group, there was no improvement on the time. Therefore we will have to rely on changing the work group size to improve on the time.

Table 5

Kernel	*accelerate_flow*	*propogate*	*av_velocity*
Time taken (s)	0.000036	0.000320	0.000018
Percentage	**9.6%**	**85.3%**	**5.1%**

5.5 Using Different Work Group Sizes

The different run times of different workgroup sizes were tested on the program for the 128x128 grid (for convenience). The results are shown below:

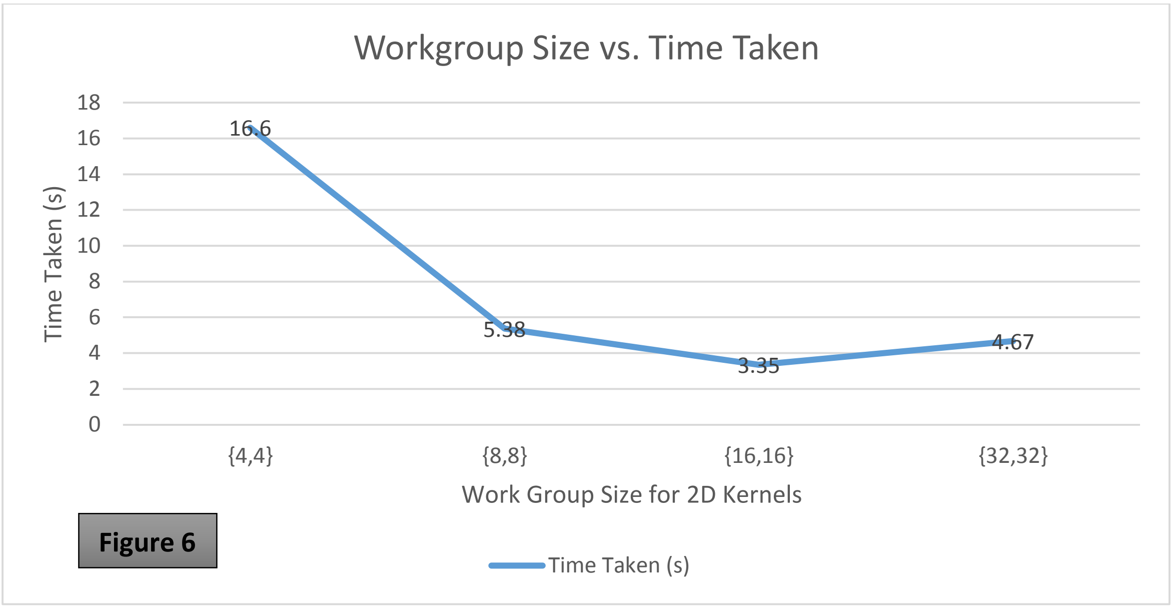

Figure 6

Therefore we choose {**16**} and {**16**,**16**} as the local workgroup size (for one dimensional and two dimensional kernels respectively) since this is the most efficient multiplier of the warp. We get **25.2s** for the **1024x1024** grid now. It should also be noted that {**1**,**1**} took a total of

6.0 Comparison of Results for Different Grid Sizes

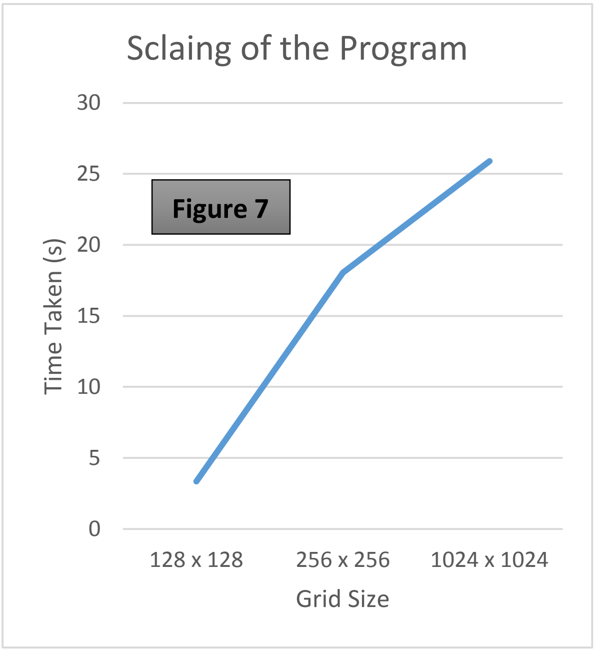

Figure 7

It must be noted that the 700x500 case clocks in at **68.16s**.

6.0 Conclusion

The best achieved for this program was **25.9s**. Through my investigations it can be seen that MPI performs the best out of OpenMP, MPI and OpenCL. However there are many optimizations that I have yet to explore from OpenCL and from general consensus, OpenCL is considered to be the most efficient. The previous section also shows us that the program scales pretty well too. The essential difference between OpenCL and the other two languages is that OpenCL uses both CPUs and GPUs. This is a field that is thriving currently and has a potential for future.