OpenCL Parallelism

Ainsley Rutterford
ar16478@my.bristol.ac.uk

I. Introduction

This report will first explore the parallelisation of the Lattice Boltzmann skeleton code via OpenCL. Later, the performance of the OpenCL implementation will be compared with Serial, OpenMP, and MPI implementations of the same Lattice Boltzmann code.

II. OpenCL

The `rebound`, `collision`, and `av_velocity` functions were ported to kernels in order to achieve a simple OpenCL implementation of the Lattice Boltzmann code. This implementation was then iteratively improved upon, ensuring that each iteration was working correctly.

The initial implementation was by no means efficient, yielding worse performance than the serial implementation for all image sizes. The inefficiencies of the implementation, and approaches to removing them, are discussed throughout the report. The runtimes are shown in Table II. Porting the `av_velocity` function to a kernel required an implementation of a reduction. The initial reduction and improvements to it are discussed in Section IV.

III. Data Movement

Initially, the code was copying the `cells` and `tmp_cells` arrays to and from the compute device each timestep. The arrays could instead be copied to the compute device once before the timesteps, and copied back afterwards.

Removing these repeated copies to and from the compute device resulted in no significant speed up for the 128x128 image, a 1.15 times speed up for the 256x256 image, and a 1.3 times speed up for the 1024x1024 image as seen in Table II. The greater increase in performance for larger image sizes highlights the fact that movement of large data sets to and from the device is costly, whereas the movement of small data sets has little to no impact on performance.

IV. Local Reductions

In order to implement a reduction in the `av_velocity` kernel, an array of floats was created on the host to store the total velocities calculated for each work-group. The array contains one float for each work-group and so the amount of memory allocated for the array was calculated by dividing the total number of cells by the size of each work-group. Thus,

$$\frac{(nx * ny)}{(local\_size\_x * local\_size\_y)} * (sizeof(float))$$

bytes were allocated for the array, where nx and ny were the dimensions of the image, and $local\_size\_x$ and $local\_size\_y$ are the work-group dimensions that will be discussed in Section IX. A buffer of the same size was created on the global memory of the compute device using the `clCreateBuffer` function.

An array of floats was also allocated in the local memory of each work-group on the compute device. These arrays would store the velocities of each work-item in each work-group and so would each be $(local\_size\_x * local\_size\_y) * (sizeof(float))$ bytes large.

The initial reduction implemented was inefficient as it only made use of one work-item per work-group. This work-item would loop through all of the local velocities and would place the sum of these velocities into the work-group's corresponding index in the global buffer. The host would then read from the global buffer to the host side array, and the array was then reduced into one `av_vels` value for each timestep.

An improved reduction on the compute device allows for more of the work-items to contribute to the reduction at once. Figure 1 shows an illustration of the reduction. This binary tree reduction allows for half of the work-items in a work-group to be contributing to the first stage of the reduction, and half of those to the next stage, and so on.

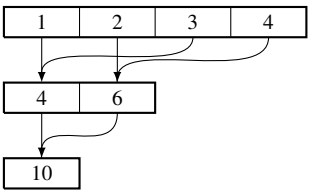

Fig. 1: A simple illustration of the parallel reduction. Reducing an array of size n requires $n - 1$ addition operations. At the end of the reduction, the result is stored in the element at index 0.

Implementation of the improved local reduction yielded a 5.7, 4.4, and 4.2 times increase in performance for the 128x128, 256x256, and 1024x1024 images respectively, as seen in Table II.

V. Data Layout

It is critical to arrange the data for each cell in an arrangement that enables memory coalescing. Memory coalescing refers to the combination of multiple memory accesses into a single transaction [1] and is often regarded as one of the most important considerations when programming for performance with GPUs [2].

The speeds of each cell were originally stored in an Array of Structures layout. This layout was non-optimal as it prevented any coalesced memory accesses. Refactoring the code to store the speeds in a Structure of Arrays arrangement ensured that the arrays of speeds were contiguous in memory, as opposed to the different speeds being 'interleaved'. Once the speeds were contiguous in memory, the GPU could perform memory coalescing.

Rearranging the data in this way resulted in a 1.5 times speed up for the 128x128 image. The increase in performance was more significant for larger images, with the 256x256 and 1024x1024 image performances increasing by 3.1 and 5.1 times respectively. This can be expected as the memory

accesses comprise a greater proportion of the runtimes for larger images, and so optimising these memory accesses results in a greater increase in performance.

VI. Kernel Fusion

The skeleton code provided iterates over the grid of cells multiple times per timestep. Each of the kernels, `propagate`, `rebound`, `collision`, and `av_velocity` iterate over the grid of cells separately.

This required multiple loads of the `tmp_cells` and `cells` arrays from the global memory of the device per timestep. As the Lattice Boltzmann code is memory bandwidth bound (as seen from the roofline analysis in Section XIII), loading of data is costly. These unnecessary loads could be avoided by 'fusing' the loops into one timestep kernel, allowing only one load of the `tmp_cells` and `cells` arrays from the device's global memory per timestep.

Once the kernels were fused, the speed values were only copied from the `cells` array to the `tmp_cells` array once per iteration. The result of the computation was now stored in the `tmp_cells` array at the end of the iteration. Before the next iteration, the pointers to the `tmp_cells` and `cells` array are 'swapped' as the correct values are now stored in alternating arrays at the end of each iteration.

Fusing the kernels resulted in a 1.9, 2.3, and 2.4 times speedup for the 128x128, 256x256, and 1024x1024 images respectively.

VII. Branching

The following excerpt from the Nvidia OpenCL Best Practices Guide [2] explains why branching can have such a large impact on the performance of GPUs: "Any flow control instruction (`if`, `switch`, `do`, `for`, `while`) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path."

Masking can be used to avoid branching. Rather than using flow control instructions to evaluate expressions only when a condition is met, the expression can always be evaluated and the result can be masked. For example,

```
if (obstacles[i + j*nx]) tot_u = 0;
else tot_u = sqrt(u_sq);
```

where `obstacles[i + j*nx]` will be either a one or a zero, can be replaced with

```
int mask = obstacles([i + j*nx]);
tot_u = mask * sqrt(u_sq);
```

to achieve the same result. This allows each work item to follow the same execution path, allowing the path to be executed in parallel, rather than being executed in serial as mentioned earlier.

Application of this technique along with other masking techniques yielded roughly a 1.15 times increase in performance for all image sizes.

VIII. Further Optimisations

Further improvements were made through the application of three minor optimisations. The first was removing the multiple `clFinish` calls. Only one was required after all computation and data transfers were complete.

The `accelerate_flow` kernel could be fused with the overall `timestep` kernel more efficiently. Instead of accelerating the flow at the start of each timestep, the flow could be accelerated during the rebound, propagate, and collision steps. The result is always computed, and is masked to only be added to the final `tmp_cells` speeds if the current row was the second row of the grid.

The final optimisation was the removal of the `tot_cells` reduction. Only the `tot_u` reduction was necessary. The total cells could be counted in the initialise function as it is already known how many cells will be blocked from the obstacles file.

Application of these three optimisations resulted in a 2 times speedup for the 128x128 image, a 1.3 times for the 256x256 image, and no significant speedup for the 1024x1024 image.

IX. Autotuning Work-group Sizes

Finding optimal work-groups sizes for the OpenCL implementation is extremely important. The Lattice Boltzmann code runs 22 times faster with the optimal work-group size than with the least optimal size.

In order to find the optimal size for this program, the Flamingo Auto-Tuning tool [3] was used. Flamingo varied the x and y dimensions of the work-groups automatically and found that the optimal sizes for the Nvidia K20m were an x size of 32, and a y size of 4. The least optimal work-group size was 1 by 1 with a runtime of 16.78 seconds for the 128x128 image. The results are shown in Figure 2. It is interesting to see that the work-group of size 32 by 4 outperforms the work-group of size 4 by 32. This is due to the image being stored in a row-major order. The rows are contiguous in memory and so the 32 by 4 work-group utilises memory coalescing more than the other work-group.

Flamingo was also used to find the optimal work-group sizes for the OpenCL implementation running on the Intel Xeon E5-2670 CPU. The optimal work-group size was found to be an x size of 128, and a y size of 2.

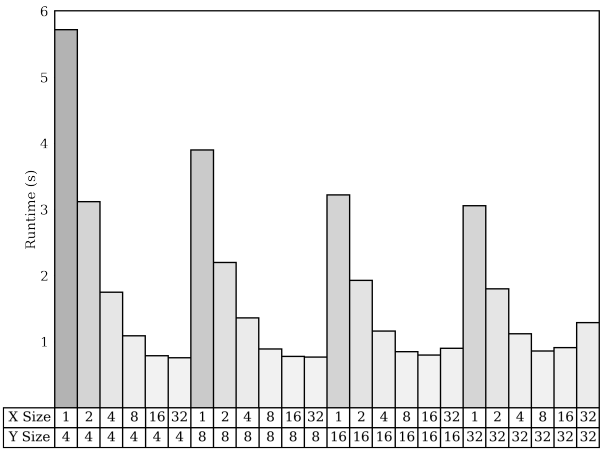

Fig. 2: The runtimes achieved for the 128x128 image by the OpenCL implementation running on a K20m with various work-group sizes. The fastest time of 0.76 seconds was achieved with an x size of 32 and a y size of 4.

X. MPI

Although this assignment focuses on parallelism with OpenCL, there are other ways to parallelise applications. While OpenCL and OpenMP support shared memory parallelism, MPI supports distributed memory parallelism. I thought it would be interesting to briefly compare the performance of the OpenCL and OpenMP implementations with an MPI implementation.

The optimised serial code was used as a starting point for the MPI implementation. Since the `cells` and `tmp_cells` arrays are stored in a row-major order, it made sense to decompose the image into horizontal strips, with halo exchanges of the top and bottom rows each timestep. Rows are already contiguous in memory and so can be sent without extra computation. Whereas, columns have to be inefficiently accessed and packed in order to be sent, and unpacked once received.

A simple communication pattern was implemented using the `MPI_Sendrecv` function. Each process would first send to the left and receive from the right, before sending to the right and receiving from the left. This pattern ensured the implementation was deadlock free. After the timesteps were complete, each process contained a local array of average velocities. These arrays were reduced to one global array by the master process using the `MPI_Reduce` function with the `MPI_SUM` argument.

The runtimes achieved by the MPI implementation are shown in Table III. Although the MPI implementation did outperform both the Serial implementation and the OpenCL implementation on a CPU, it is worth noting that it could have been improved upon in various ways (for example via an asynchronous communication pattern, or domain decomposition via a grid).

XI. OpenMP 4.5 Target Off-loading

An OpenMP implementation using the `target` pragma was also explored. The `target enter data map(to: ...)` pragma was used to offload the `tmp_cells`, `cells`, and `obstacles` arrays to the compute device. The `target exit data map(from: ...)` pragma was used to read the `cells` values back to the host again.

The `target teams distribute parallel for collapse(2) reduction(+:tot_u)` pragma was used to parallelise the main nested `for` loops in the `timestep` function. The `teams distribute` keywords specify that iterations will be shared among the master threads of all thread 'teams' in the 'league' created by the `teams` construct [4]. The `collapse(2)` clause collapses the nested loops into one. If the loops were not collapsed, only the outer level could be parallelised.

The `reduction(+:)` clause generates a summation reduction on the variable specified. The results produced by this simple OpenMP Target implementation were underwhelming. A runtime of 13.15 seconds was achieved for the 128x128 image. It is worth noting that the reduction generated by OpenMP is likely not efficient. This can be seen when comparing the BabelStream [5] runtimes achieved by OpenMP and OpenCL in Table I. The 'Dot' kernel is the only kernel that contains a reduction. OpenMP achieves similar results for all kernels, but performs significantly worse for the 'Dot' kernel, suggesting that the reduction generated by OpenMP is not efficient.

TABLE I: Runtimes achieved by OpenCL and OpenMP 4.5 running the BabelStream benchmark on the Nvidia K20m.

	OpenCL (s)	OpenMP 4.5 (s)	Difference
Copy	0.00358	0.00368	2.7%
Mul	0.00360	0.00366	1.6%
Add	0.00533	0.00546	2.4%
Triad	0.00533	0.00567	5.9%
Dot	0.00371	0.01061	65.0%

XII. Choice of Compilers and Flags

Throughout development, various versions of both the Gnu, and Intel compilers were compared. Ultimately, the Intel 16.0.2 compiler with Gnu 7.1.0 compatibility, in conjunction with the `-Ofast`, and `-xHOST` flags, provided the fastest times for the OpenCL implementation. The previously written Serial and OpenMP implementations use the `_mm_malloc` function and are therefore also compiled using the Intel 16.0.2 compiler with Gnu 7.1.0 compatibility.

The MPI implementation was compiled with the Intel MPI Library 5.1.3 with the Intel 16.0.2 compiler with Gnu 7.1.0 compatibility for the sake of consistency. The `-Ofast`, and `-xHOST` flags were used once again.

The OpenMP 4.5 Target implementation was compiled using the Clang compiler with the relevant flags needed to target a Nvidia GPU. The flags can be found in the Makefile submitted.

It is also possible to use flags when compiling the kernel programs using the `clBuildProgram` function. The `-cl-fast-relaxed-math` flag was experimented with, as well as the `-cl-mad-enable`, flag which allowed the use of the `mad` function. The `mad` function can be used to calculate `a * b + c` more efficiently but with reduced accuracy [6]. However, when experimenting with these flags, there was no significant difference in performance.

TABLE II: Runtime in seconds for each image size for the various iterations of the OpenCL implementation. The code was compiled using the compiler and flags outlined in Section XII.

	128	256	1024
Initial OpenCL	37.09	252.79	939.01
Improved Data Movement	37.12	222.10	727.17
Improved Local Reduction	6.53	51.85	172.49
Coalesced Memory Access	4.36	16.86	34.96
Kernel Fusion	1.62	6.29	15.12
Masking	1.55	5.49	13.22
Further Optimisations	0.76	4.18	13.32

TABLE III: Runtime in seconds for each image size for the various implementations of the Lattice Boltzmann Code. Each implementation was compiled using the relevant compiler and flags outlined in Section XII.

	128	256	1024
Serial	6.33	53.28	256.32
OpenCL CPU	2.98	19.15	90.92
MPI	1.92	14.11	77.84
OpenMP	0.79	4.41	39.25
OpenCL GPU	0.76	4.18	13.32

XIII. Results and Roofline Analysis

It is important to know the maximum achievable performance of a system in order to know when further optimisations are either no longer worthwhile, or even possible.

The STREAM benchmark [7] can be used to measure the maximum bandwidth of which a system is capable. The peak bandwidth for the Nvidia K20m Kepler is 151 GB/s. The bandwidth achieved by the Lattice Boltzmann program is calculated by dividing the total memory usage by the runtime achieved. Thus, the bandwidth achieved is calculated as follows:

$$\frac{size^2 * nspeeds * sizeof(float) * 2 * iterations}{runtime}$$

The factor of 2 in the above equation reflects the fact that the cells are read from and written to, each iteration. For the 1024x1024 image, a bandwidth of $1024 * 1024 * 9 * 4 * 2 * 20000/13.32 \approx 113$GB/s is achieved, and the fraction of STREAM bandwidth achieved is $(113/151 \approx 75\%)$. The bandwidths achieved by each implementation are shown in Table IV.

The operational intensity of the program is calculated by dividing the number of floating point operations by the number of bytes loaded or stored. The Serial, OpenMP, and MPI implementations all had 122 FLOPs per iteration, and 18 floats were loaded or stored (9 `cells` speeds are loaded, 9 `tmp_cells` speeds are stored). Thus, the Operational Intensity of these implementations is $122/(18 * 4) \approx 1.69$ FLOP/Byte. The OpenCL implementation involved extensive masking, as discussed in Section VII. Due to these extra FLOPs, the OI calculated for the OpenCL implementation is $155/(18 * 4) \approx 2.15$.

Multiplying the bandwidth by the OI results in the Performance (GFLOP/s) achieved. The roofline graph containing the performance of the OpenCL implementation on the GPU is shown in Figure 3. The roofline containing the implementations running on the CPU is shown in Figure 4.

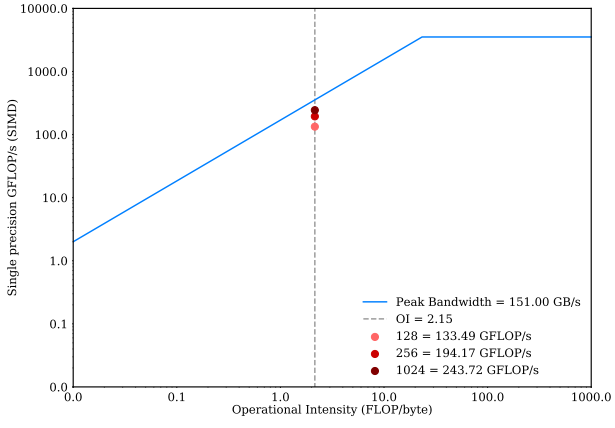

Fig. 3: The Roofline model of the Nvidia K20m Kepler GPU. The performance achieved by the OpenCL implementation for all image sizes is shown. The OpenCL implementation has an OI less than 23.3, and so is memory bandwidth bound.

XIV. Implementation Comparisons

One benefit of using the OpenCL framework is that it enables heterogeneous computing. The code developed in this assignment can be run on CPUs as well as GPUs. However, when running the OpenCL code on the Intel Xeon E5-2670 CPU, the runtimes achieved were slower than those achieved by both the OpenMP and MPI implementations.

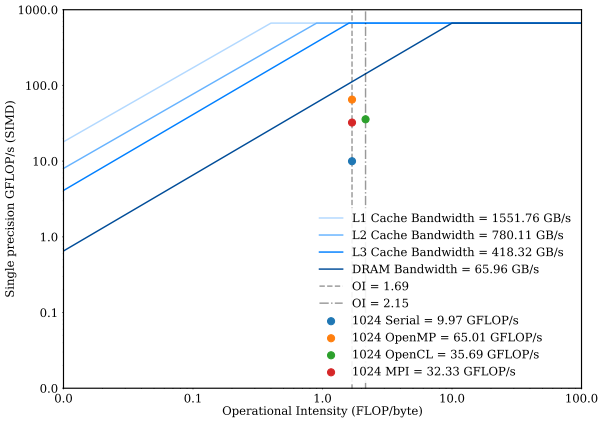

Fig. 4: The Roofline model of Intel Xeon CPU E5-2670 (Sandy Bridge), 16 cores. For simplicity, only the 1024x1024 image performances are shown.

TABLE IV: The Bandwidths, Performances and Operational Intensities achieved by each implementation for the 1024x1024 image. The Bandwidths are in GB/s, Performances in GFLOP/s, and Operational Intensities in FLOP/byte.

	Bandwidth	Performance	OI
Serial	5.90	9.97	1.69
OpenCL CPU	16.60	35.69	2.15
MPI	19.13	32.33	1.69
OpenMP	38.47	65.01	1.69
OpenCL GPU	113.36	243.72	2.15

The OpenCL implementation running on the Nvidia K20m achieved the greatest bandwidth of any implementation. Although the peak bandwidth of the Intel Xeon E5-2670 CPU (seen in Figure 4) is far greater than that of the GPU, this bandwidth requires use of the L1 cache, which is extremely small. Though implementations running on the CPU may provide comparable performance for smaller image sizes, they do not scale well. This can already be seen when comparing the OpenMP 1024 runtime with the OpenCL GPU 1024 runtime in Table III. This can be expected, as the Intel Xeon's peak DRAM bandwidth is only 65 GB/s, compared to the K20m's 151 GB/s.

References

[1] Introduction to GPGPU and CUDA Programming: Memory Coalescing. https://cvw.cac.cornell.edu/gpu/coalesced.

[2] NVIDIA OpenCL Best Practices Guide Version 1.0. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[3] Flamingo Auto-Tuning. http://mistymountain.co.uk/flamingo/.

[4] Jeff Larkin. OpenMP and NVIDIA. https://openmp.org/wp-content/uploads/SC13_OpenMP_and_NVIDIA.pdf.

[5] Deakin T, Price J, Martineau M, and McIntosh-Smith S. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models., 2016.

[6] Aaftab Munshi. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[7] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.