Parallel Optimization of Lattice Boltzmann Code Using OpenCL

Harry Waugh - hw16471

***Abstract*—This report explores the efficiency of different OpenCL optimization techniques on a skeleton Lattice Boltzmann code. It also evaluates CPU performance against other programming platforms such as MPI and OpenMP. Whilst OpenCL GPU performance is tested on different compute devices, and uses both single and multiple GPU configurations. A summary of each implementation's performance can be seen in Figure 1.**

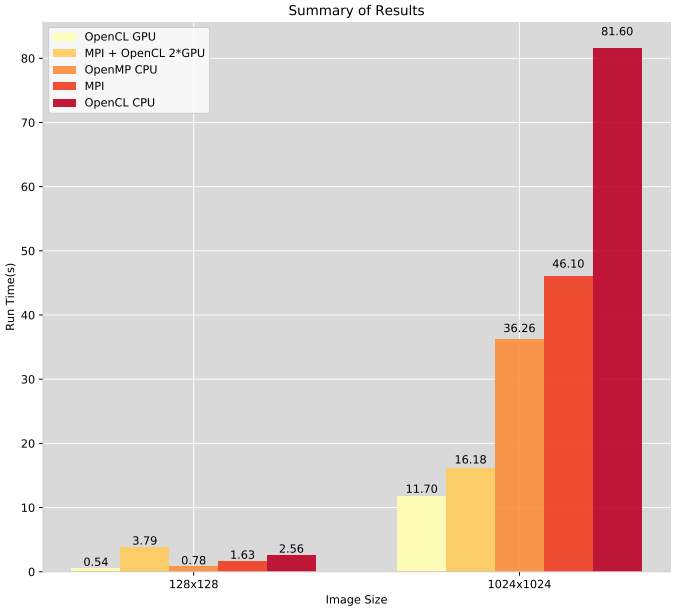

Figure 1: Summary of Results.

I. Compiler Flags

When optimizing a given piece of code, it is common to first compile the code with different compilers and various flags. This optimization requires no code manipulation and can dramatically improve performance. To test the impact of different compilers, a shell script was used to compile the Lattice Boltzmann (LBM) code on different GNU and Intel Compilers along with different combinations of compiler flags. The results are shown in Table I. These were obtained by using the most recent GNU compiler (GCC 7.1.0) in conjunction with the `-Ofast`, and `-xAVX`. By using this compiler and combination of flags, speedups of 1.7x were achieved across all image sizes.

II. Kernels and Reductions

Using the NVIDIA Visual profiler [1], the proportion of time that the graphics card spends performing operations such as transferring to and from memory and running kernels can be identified. By examining this data, it can be seen that a major bottleneck is encountered at each timestep when the cell arrays are copied from the host memory to the GPU global memory and then back again. A solution to this is to compute each timestep entirely using memory on the GPU. This was implemented using separate kernels for each of the `rebound()`, `collision()` and `average_vels()` functions.

One consideration when writing the kernel for the `average_vels()` function was how to efficiently compute the sum of the total cells and total speeds, which is done using a reduction found in [2].

This works by each work item calculating the speed of its cell and then storing this speed in an array stored in local memory, which is shared by all items in the work-group. Using memory barriers to synchronize the work items, they then sum the local array in parallel, where alternate work items add the next consecutive work items speed value. This is repeated $log_2(workitems)$ times until the total speed of the work group is in the 0th position of the local array. A single work item then stores the total work group speed in the work groups unique position in the global total speed array, this is then summed back on the host CPU. An illustration of this parallel tree reduction can be seen in Figure 2.

By porting these functions to kernels, the amount of memory that is transferred from the GPU back to the host per timestep is reduced significantly. Assuming a 16x16 work group size, the 128 image transfers 1 float for each work group (`64*sizeof(float) = 256 bytes`) back to the host, which is far less than the size of the cells array (`128*128*nspeeds*sizeof(float) = 576 kilobytes`). These improvements translated to large speedups across all image sizes. The 128 image time was improved the least with a speedup of 6.75x, but for the larger images with greater cell array sizes, speedups were up to 20x faster.

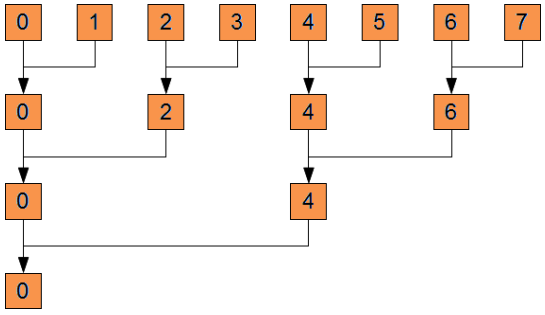

Figure 2: Parallel Reduction.

III. Kernel Fusion

At this stage the information flow of the LBM code during each timestep starts in the GPU cell array, moves to the temporary cells array in the `propagate()` kernel, and then is moved back to the cells array during the `collision()` or `rebound()` kernels. To remove this needless transfer of memory these three kernels can be fused together, and the `propagate()` memory transfer can be merged into the other functions. To ensure that the correct cells array is operated on during the next timestep a pointer swap was implemented

here.

An advantage of fusing these three kernels into a single `timestep()` kernel is that the cells array is loaded once, operated on, and then stored once at the end of the kernel. There is also a set of redundant memory loads and stores in the `accelerate_flow()` kernel. This can be computed at the end of the new `timestep()` kernel, preventing memory from being loaded and stored again. Fusing the kernels yielded a 2x speedup for the 128 image, whilst the performance of the other two images were improved by 1.2x.

IV. Coalesced Memory Access

A coalesced memory access is the optimum memory access pattern for a GPU [3]. Ensuring that consecutive work items work on consecutive parts of memory should help enable coalesced memory accesses. This allows a work group to access multiple elements with a single transaction.

In the case of the LBM code, coalesced memory access occurs when the cells array is in a structure of arrays form rather than an array of structures form. After refactoring the LBM code to store the cell speeds in a structure of arrays, significant speedups were produced across all problem sizes. The problem size appeared to be correlated to the performance for this optimization. The 128 size performed 1.3x faster and the larger 1024 size performed over 2x faster.

V. Branch Divergence

At this step in the OpenCL optimization process, the main `timestep()` kernel features several branches that perform different computations and store different values in memory based on whether there is an obstacle or not. This presents a greater issue when programming GPUs in comparison to CPUs, leading to branch divergence.

Divergent branches occur on a GPU because of it's Single-Instruction-Multiple-Data (SIMD) architecture. In an optimal program, all work items in a work group will follow the same control flow path through the kernel. However, divergent branches conflict with this and mean that branches could be executed after each other, rather than in parallel.

A common solution to this problem is to use ternary operators to initialize mask values of 0 or 1. The mask can then be multiplied with different branch outputs in order to select which values are stored to memory. This technique was used to choose which of the `collision()` or `rebound()` outputs were stored in memory, along with when to compute the `accelerate_flow()` function.

Refactoring the `timestep()` kernel to limit these divergent branches decreased the run times for each problem size. The two smaller images 128 and 256 produced speedups of 2x and the 1024 image had a speedup of 1.4x.

VI. Global Reduction

Section II outlines a reduction method in which the average velocity calculated by each work group per timestep is passed back to the host and summed before the next `timestep()` kernel. Although this is a quicker alternative than sending back the whole of the cells array, there is room for improvement.

This improvement was achieved by completing the reduction as before, and storing all the work group's average velocity each timestep. Then after completing all iterations, a `reduce()` kernel was used to individually sum each iteration's work group totals, ready to be sent back to the host. This provided the greatest speedup for the smaller problem size 128, which had a speedup of 1.6x. The speedup for the 256 and 1024 sizes were 1.25x and 1.08x respectively. One reason for this is that the larger problem sizes spend a higher percentage of time computing the timestep function, and overheads like reading to and from buffers make up a lower proportion of their run time.

VII. Optimizing Work Group Sizes

Until this point an arbitrary work size was chosen to be 16x16 for the size of each work group. After optimizing the code, an experiment was run for each problem size to identify performance 'sweet spots'. The Flamingo Autotune tool [4] was used to test each work group size $S \times S$, where $S \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. It is obvious from the results in Figure 3 that the LBM code on a Tesla K20m suits work group sizes that have a low Y dimension and a much higher X dimension. This trend was consistent for all problem sizes because the rows are stored contiguously in memory, suiting coalesced memory access. The blue, purple and black circle outlines indicate the fastest times for each of the 128, 256 and 1024 sizes respectively. The 32x8 work group size better suited larger images and so was used onward. Using this size, speedups across each of the 128, 256 and 1024 problem sizes were 1.05x, 1.1x and 1.06x.

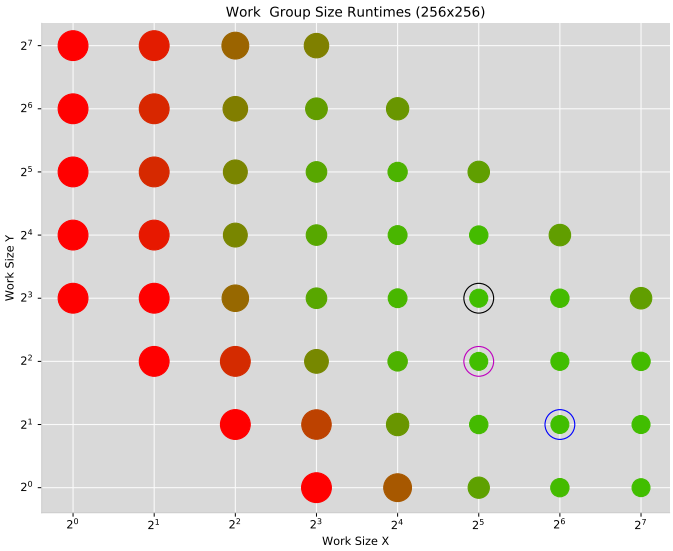

Figure 3: Run times for different work group sizes on the 256 problem size. Small and bright green circles represent faster times, large and red circles represent slower times.

VIII. OpenCL Performance

A measure of OpenCL performance is given using the roofline model shown in Figure 4. The operational intensity (OI) was calculated to be 1.82 ($\approx 131/72$) and is the amount of floating point operations per byte of memory access. This OI indicates that the LBM code is bound by the peak memory-bandwidth of an Nvidia Tesla K20.

The fraction of STREAM bandwidth for each data size can be calculated by first working out the bandwidths achieved. This is done by summing the total number of bytes that have been accessed from memory, and then dividing this figure by the run time. For example, the 1024 size accesses a total of 72MB per iteration, and therefore has a memory bandwidth of 129GB/s when completing 20,000 iterations in 11.7 seconds. The 128 achieved a memory bandwidth of 87GB/s and the

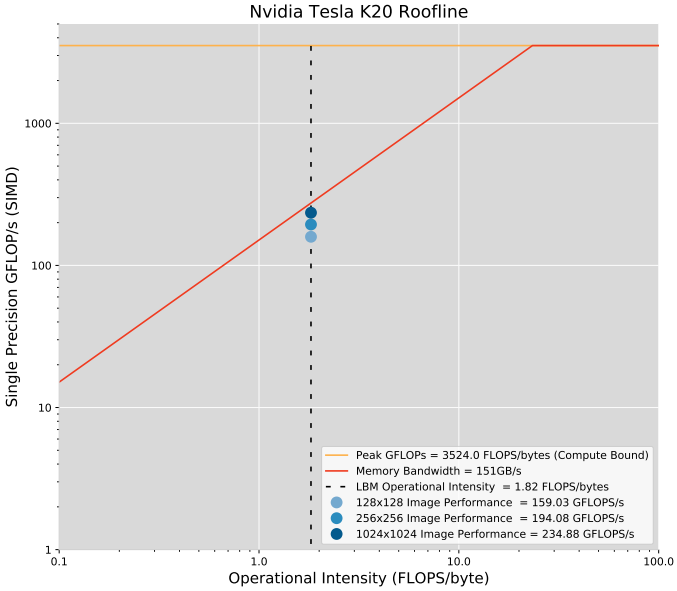

Figure 4: Roofline Analysis of OpenCL Performance using a Nvidia K20 GPU.

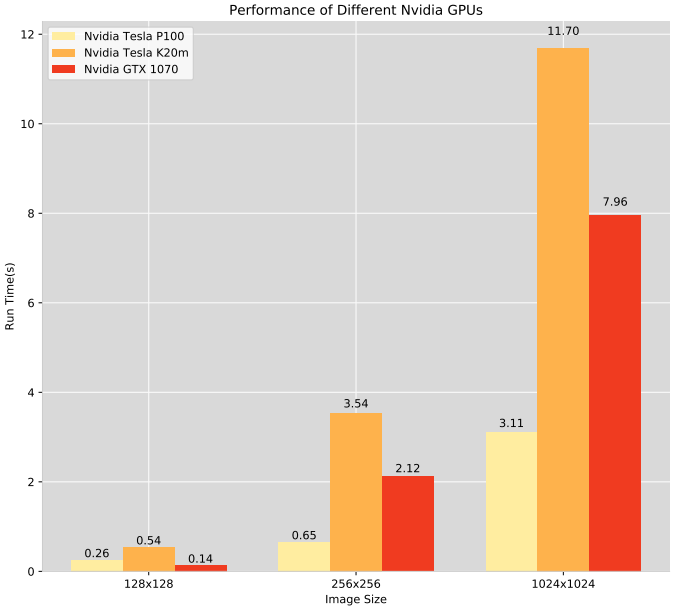

Figure 5: Comparison of LBM run times using different GPUs.

256 achieved 106.6GB/s. Therefore the achieved fractions of STREAM bandwidth for the 128, 256 and 1024 problem sizes were 57.6%, 70.5% and 85.4% respectively. Crucially this shows that as the input size increases, the memory bandwidth achieved becomes closer to the peak of the underlying hardware.

IX. GPU Performance Portability

A key advantage of OpenCL over MPI and OpenMP is its heterogeneous nature and ability to run on different hardware such as CPUs and GPUs. Because of this, an important aspect of OpenCL is how portable its performance is across different platforms. To keep emphasis on optimizations themselves, all times mentioned so far were measured using an Nvidia Tesla K20 (K20). Performance on this was then tested against two other Nvidia GPUs, a Tesla P100 (P100) and an Nvidia GTX 1070 (1070). The results are shown in Figure 5.

As seen in Figure 4, the LBM code is clearly memory-bandwidth-bound on a K20 and achieves up to 85.4% of the peak STREAM bandwidth. Therefore it would be expected that a GPU with greater memory-bandwidth such as 256GB/s for the 1070 or 732GB/s for the P100 would achieve much better performance. The P100 for example, has 3.5x the peak memory-bandwidth of the K20, and has a run-time that is 3.76x faster. It is also interesting to see that the P100 is the quickest of all the GPUs on both the larger data sizes, but is slower than the 1070 on the 128 size. It is unclear why this is the case, as the 1070 has less memory-bandwidth. A possible reason for this is because there is lower proportion of memory movement due to the smaller arrays. Therefore, the 1070's quicker clock speed may make an impact.

X. Implementations for OpenMP and MPI

This section outlines the optimizations that were undertaken during the implementations for OpenMP and MPI, so that their performance can be compared to the OpenCL performance when running on a CPU in Section XI.

Both OpenMP and MPI implementations were first serially optimized. This meant expressions were factorized and loops were fused, allowing the data to be loaded from memory once, operated on, and then stored back into memory. Intel also suggests that data is stored in a structure of arrays format and aligned to cache lines to help ensure vectorisation on the CPU [5].

OpenMP specific optimizations included using pragma clauses such as `parallel`, which forked threads so that they shared the computational load. The `reduction` clause was also used to create an aggregate of each thread's total speed. Cell data was initialized with awareness of Non-Uniform Memory Access. This ensured that each thread allocated the cell data in a location that was physically close to them.

The MPI implementation was more complex to implement and involved explicitly exchanging halos of border data between neighbouring processes at each iteration of the loop. To reduce the communication overheads when transferring halos, the `MPI_Type_vector` was used to define the stride between the halos of each speed array. This reduced overheads as 9 halo transfers could be done together. The `MPI_Reduce` function was used after all iterations were complete to sum all of the processes' individual iteration totals.

XI. Comparing CPU Performance

The OpenCL code described in the first half of this report can be executed on both GPUs and CPUs. This section evaluates the performance of the code on a CPU and compares it to the MPI and OpenMP implementations described in Section X.

The results in Table I show that the OpenCL CPU code consistently achieved slower run times than when running on a GPU. However, this is to be expected because the LBM code is memory-bandwidth-bound, and the STREAM memory-bandwidth of the CPU is only 66GB/s compared with the GPU's 151GB/s. Another performance limiting factor here is that the CPU can only simulate the local memory that the GPU has. It is important to note that the CPU and GPU work group size differs, a 128x8 size was found to be optimal for the CPU.

When tested against OpenMP and MPI implementations, OpenCL consistently had slower run times. To measure the performance of each of these implementations, the roofline model was used to identify the achieved fraction of STREAM

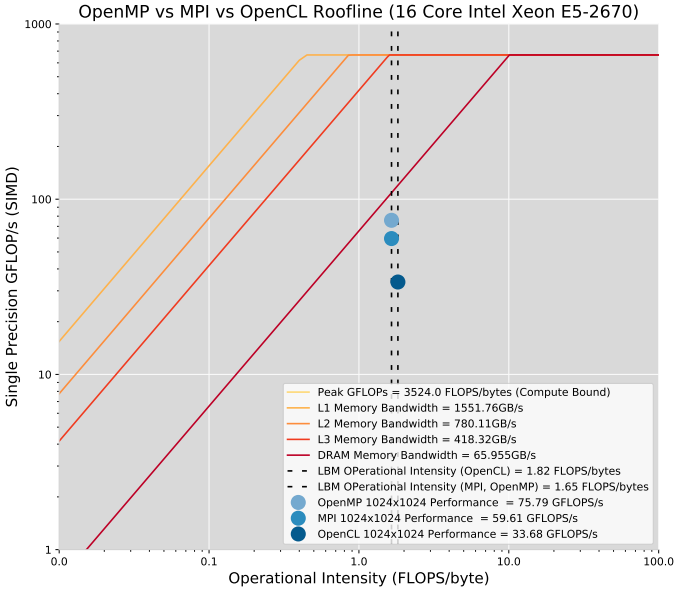

Figure 6: Comparison of CPU LBM performance between MPI, OpenMP and OpenCL Implementation using the 1024x1024 data size.

bandwidth for two Intel Xeon E5-2670 CPUs. A graph of the 1024x1024 roofline is shown in Figure 6.

The operational intensity (OI) for the OpenCL LBM code was calculated to be 1.82 ($\approx 131/72$) by identifying the number of FLOPs per byte of memory movement. The OI differed for the OpenMP and MPI implementations as the `accelerate_flow()` loop and `timestep()` loop were kept separate.

The amount of memory bandwidth achieved was calculated by multiplying the size of the speed arrays by the number of iterations, and then dividing by the run time ($72MB * 20000/runtime$). The memory bandwidth achieved by the OpenMP, MPI and OpenCL implementations were 40GB/s 31GB/s and 18GB/s respectively.

Therwfore, the STREAM bandwidth achieved can be calculated by dividing the STREAM DRAM bandwidth by the achieved memory bandwidths. The DRAM bandwidth is used because the two CPUs only have 20MB of L3 cache each, and therefore a portion of the 72MB of speed arrays must be stored in main memory. Using this value, the obtained fraction of STREAM bandwidth for each implementation is (OpenMP: 60%, MPI: 47%, OpenCL: 18%).

Both the OpenMP and MPI implementations achieve around half of the STREAM bandwidth, with the OpenMP achieving the best performance on a CPU. The OpenCL code achieves only 18% of the STREAM bandwidth, which is significantly less than the others. However, the OpenCL code was designed for GPU performance and it is important to note that the code is still over 25x faster than the skeleton code for the 1024 problem size.

XII. MPI + OpenCL

From Sections VIII and XI it can be seen that the achieved memory-bandwidth is far greater when using OpenCL on a GPU than any CPU implementation. This is because the LBM code is memory-bandwidth bound, and a GPU has faster memory-bandwidth. This section explores the use of using 2 Nvidia Tesla K20 GPUs in an attempt to further improve performance.

This is done by combining MPI and OpenCL, and works by spawning two CPU processes that first split their speed arrays in half, and then write to their own GPUs buffer. In between running `timestep()` kernels on their respective halves, they each read their own calculated halos from their buffers, and then exchange these with each other before writing back them to their own buffers. `MPI_Reduce()` is used after all iterations are complete to get the aggregate total speeds for each iteration.

Unfortunately this implementation does not perform any faster for any of the given data sizes. Through profiling, the root of this issue was identified as the large amount of communication that occurs in order to exchange halos. This is because halos must first be read from the GPU buffers, before being sent to the other process to be written back again, all of which are computationally expensive. The times achieved on the given data sizes are shown in Table I. It can be seen that times were achieved for the 1024 problem size were comparable to those achieved using a single GPU. However, the 128 size becomes 7x slower.

An experiment was also run on a larger 2048x2048 problem size to see if a greater computation to memory transfer ratio could make use of the extra GPU. The results showed that the MPI+OpenCL code was significantly faster, achieving a time of 34.3 seconds compared with the single GPU time of 45.4 seconds, representing a speedup of 1.32x.

XIII. Conclusion

After comparing various different LBM implementations that runs on CPUs and GPUs, it is clear that the the GPUs' larger memory-bandwidth allows it to outperform any CPU implementation. Furthermore, this report shows that the implementation that has the best performance for problem sizes below 1024 is OpenCL with one GPU. However, using MPI+OpenCL with two GPUs is faster for larger problem sizes.

Table I: Final Results

Optimization Technique	**128**	**256**	**1024**
OpenCL GPU (Skeleton)	75.8	597	2150
OpenCL GPU (Compiler Optimizations)	44.0	325	1100
OpenCL GPU (Kernels and Reductions)	6.51	18.4	54.1
OpenCL GPU (Kernel Fusion)	3.23	14.7	42.4
OpenCL GPU (Coalesced Memory)	2.42	9.16	19.2
OpenCL GPU (Branch Divergence)	1.02	4.8	13.5
OpenCL GPU (Global Reduction)	0.62	4.00	12.5
OpenCL GPU (Optimized Work Size)	0.54	3.54	11.7
MPI + OpenCL (2*GPU)	16.9	18.3	15.5
OpenCL CPU	2.56	15.7	81.6
OpenMP CPU	0.78	4.36	36.3
MPI CPU	1.63	11.6	46.1

References

[1] "Nvidia Visual Profiler." https://https://developer.nvidia.com/nvidia-visual-profiler.
[2] R. Tay, "OpenCL Parallel Programming Development Cookbook," August , 2013.
[3] Nvidia, "Nvidia OpenCL Best Practices Guide Version 1.0."
[4] "Flamingo Autotuning Tool." https://http://mistymountain.co.uk/flamingo/.
[5] R. Krishnaiyer, "Data Alignment to Assist Vectorization." https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.