AHPC - Lattice Boltzmann OpenCL

Jerry Wang (sw16437)

I. Compiler Flags

This is a report on the optimisation of the OpenCL implementation of lattice boltzmann algorithm. The program was tested on BlueCrystal4 with the following hardware:

- Intel Xeon CPU E5-2680 v4 @ 2.40GHz
- Tesla P100-PCIE-16GB
- Work Group Size 32*4

The timing for the example code before any optimisation are listed below in Table I.

Before making any change to the code, it is worthwhile to make sure the program is compiled with the most adequate optimisation flags. By using ICC instead of GCC and adding optimisation flag `-xHOST -qopenmp`, the program saw an immediate speed improvement of 1.07x.

Size	128*128	256*256	1024*1024
Time	44.6	330.5	357.3

TABLE I
EXAMPLE OPENCL CODE RUNTIME

II. Kernel Porting

In the original code, `accelerate_flow()` and `propagate()` are already ported to kernel, but `rebound()`, `collision()` and `av_velocity()` remain in host program. As GPUs have far more cores than CPUs and are capable of running more threads in parallel, porting these functions to kernel should enable a much greater degree of parallelism. After porting the above three functions to kernel (with no reduction implemented), we can observe an increase in speed of 2.7x.

Size	128*128	256*256	1024*1024
Time (s)	14.9	80.1	262.6
Speed-Up (x)	2.9	4.1	1.3

TABLE II
RUNTIME AFTER PORTING KERNELS

III. Reducing Data Movement

The original code writes the `cells` to and from OpenCL buffer in each iteration of `timestep()` in order to calculate the lastest average velocity. This is unnecessary as both the grids and the average velocity array can remain in OpenCL buffer and does not need to be extracted until the very end of the program. In order to achieve this, a new `av_vels` array is created in OpenCL buffer to store the average velocity after each iteration. This array will only be read back to host after the last iteration, eliminating all unnecessary data movement during time steps.

Because the runtime of OpenCL programs are mainly bounded by the amount of memory movement rather than computation, it is essential to avoid transferring data to or from the GPU unless absolutely needed.

By removing the reading and writing of `cells` inside `timestep()`, we can observe an increase of 3.6x in speed.

Size	128*128	256*256	1024*1024
Time (s)	4.6	20.1	82.3
Speed-Up (x)	3.2	3.9	3.1

TABLE III
RUNTIME AFTER REMOVING EXTRA READS AND WRITES

IV. Removing clFinish()

In the host code, all functions that invoke kernels use `clFinish()` at the end to check that all work items have finished. However, `clFinish()` is not necessary between kernel calls if all kernel calls execute on the same command queue and the queue is in order. There is overhead in each `clFinish()` call that sends all queued commands to the devices and checks their execution status.

Because `clFinish()` blocks until all previously queued OpenCL commands in a command-queue are issued to the associated device and have completed, `clFinish()` calls create breaks in the work being done by the GPU as the GPU has to wait for the next kernel to be enqueued and sent to the device after

`clFinish()`. Omitting these calls to `clFinish()` means that kernel calls can be sent to the GPU while the previous kernels are still executing. By removing all but the last `clFinish()`, we can make sure all work items are still properly executed while observing a speed improvement of 1.3x.

Size	128*128	256*256	1024*1024
Time (s)	2.9	17.1	81.5
Speed-Up (x)	1.5	1.2	1.0

TABLE IV
RUNTIME AFTER REMOVING CLFINISH

V. BASIC SUM REDUCTION

In a naive `av_velocity()` kernel, the `tot_u` for each cell is calculated and stored in a separate OpenCL global arrayof size $nx * ny$ which is then read back to the host to calculate the sum. This causes significant overhead especially in larger-size grids. To improve performance, local work groups are used and the summing operation of `tot_u` is divided evenly into chunks. Each work group produces a single partial sum, and the array containing partial sums are then read back to the host. This allows for a much smaller and faster sum.

In the case of porting `av_velocity()`, `barrier(CLK_LOCAL_MEM_FENCE)` is added in order to synchronize all the work items. This resulted in a speed improvement of 1.4x.

Size	128*128	256*256	1024*1024
Time	2.2	10.6	59.9
Speed-Up (x)	1.3	1.6	1.3

TABLE V
RUNTIME AFTER IMPLEMENTING 1D REDUCTION

VI. STRUCT OF ARRAY

The original code stored the cells in array of structures. Each cell in the grid is an individual structure storing 9 speeds, which is then stored in an array. In order for the code to have a higher chance of accessing contiguous memory, the code needs to be transformed to storing cells simply in a large array of floats. Because `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` does not perform deep copy to or from kernel, separate speed arrays are not used in this implementation. Instead, all 9 speed arrays are combined into one contiguous array. This would change the access of cell `(ii, jj)` from `cells[ii + jj*nx].speeds[kk]` to `cells[(kk * nx * ny) + ii + jj*nx]`.

After applying the above step, we observe an immediate speed up of 3.2x.

Size	128*128	256*256	1024*1024
Time (s)	1.8	4.4	9.9
Speed-Up (x)	1.2	2.4	6.1

TABLE VI
RUNTIME AFTER APPLYING SOA STRUCTURE

VII. KERNEL FUSION AND POINTER SWAP

During each time step, the grid is being en-queued in multiple kernels several times. `propagate()` reads data from `cell` and writes into `tmp_cell`. `rebound()` and `collision()` read data from `tmp_cell` and write into `cell`. Repeated reading and writing creates unnecessary memory usage and produces significant overhead. Therefore, reducing the reading and writing of the grid should provide considerable performance improvement. As a result, `propagate()`, `rebound()` and `collision()` are combined into a single kernel, moving the data only once per time step from one grid to another.

With the kernels fused into one, it is important to also implement pointer swap so that the updated grid is always used as the source grid in the next iteration. However, implementing pointer swap in OpenCL is more complicated than in OpenMP, as the pointers to OpenCL memory can not be swapped directly from `main()`. This problem can be addressed by creating copies of the time step functions with opposite kernel arguments that utilise the same kernels. These functions include `accelerate_flow_out()`, `accelerate_flow_in()`, `collision_out()` and `collision_in()` which move data only once but in opposite directions.

This did not produce massive improvement but it is expected. Using GPU trace and summary in nvprof, it is evident that the lost runtime is introduced by the `if` statement inside the newly merged `collision()` function, which is used to check if an element contains any obstacle.

This is possible since branch divergence can potentially break some of the optimisation by the compiler. Because the GPU has plenty of computation power in reserve, and Lattice Boltzmann method is a memory-bounded task, it is reasonable to take advantage of

the vastly available computation power for reduced memory bandwidth usage. Thus, we can remove the `if` statements, meaning calculating each element both as an obstacle and not an obstacle, and only evaluate if it is an obstacle upon the final assignment to the destination grid using an conditional operator. This produced a decent improvement of 1.5x.

Size	128*128	256*256	1024*1024
Time	0.8	3.9	7.8
Speed-Up (x)	2.2	1.1	1.2

TABLE VII
RUNTIME AFTER APPLYING LOOP FUSION

VIII. ADVANCED SUM REDUCTION

Although basic sum reduction did reduce runtime on smaller size grids such as 128*128 and 128*256, it provided very limited benefit while processing larger grids such as 256*256 and 1024*1024. This is caused by the larger number of work groups needed from using a relatively small work group size. Because each work group produces a partial sum, larger grids will require a larger number of work groups, thus producing a significantly larger partial sum array that is expensive to be summed up, which may lead to noticeable slow down.

Using nvprof, we can observe that in the largest grid of 1024*1024, an astonishing 88% of total runtime is consumed by the loop responsible for summation. Therefore it is necessary to implement a more efficient reduction. Instead of summing the array element by element, a dynamically adjusted stride is used. This allows the length of the array to be folded in half after each iteration, reducing the time complexity of summation from $O(n)$ to $log(n)$.

By applying the new reduction to both local sums and partial sums, we see an immediate improve in performance in larger grids. However, the smallest grid 128*128 did not improve and even experienced a slight slow down. This may be that smaller grids do not require as many work groups, thus producing a smaller partial sum array, which does not benefit from using the new reduction.

IX. WORK GROUP SIZE

In OpenCL, using work-groups allows more optimization for the kernel compilers. This is because data is not transferred between work-groups, and local memory provides much faster access speed than global

Size	128*128	256*256	1024*1024
Time	0.9	1.9	4.9
Speed-Up (x)	1.0	2.1	1.6

TABLE VIII
RUNTIME AFTER IMPLEMENTING 2D REDUCTION

memory. For a fixed number of cells, a larger work group size requires fewer work groups in total, and a smaller work group size requires more work groups in total. Larger work group size takes longer to execute but produces fewer partial sums. Smaller work group size executes faster but produces a large number of partial sums. Therefore, it is important to find the right work group size to achieve the optimal performance.

Table (IX) shows the runtime of the program using different work group sizes when processing a grid size of 128*128. From the table, we can observe that a wider work group is generally faster than a narrower work group. This is possibly because a wider work group has higher chance of accessing contiguous memory. A work group size of 64 also seems more likely to produce the fastest outcome. This is likely because the Tesla P100 GPU has a SIMD width of 64. It is also worth noting that the largest work group size 64*64 prompt an error when the kernels are enqueued. This may be a limit of the number of registers. As each work group requires a certain number of registers for local memory access while the total number of register is limited. If the size of the workgroup is too big, there may not be enough registers available, thus prompting an enqueue kernel error.

Finally for a size of 128*128, work group size of 32*2 produces the fastest runtime.

Runtime	X=2	X=4	X=8	X=16	X=32	X=64
Y=2	2.07	1.81	1.48	1.4	1.08	1.14
Y=4	1.65	1.57	1.22	1.18	1.2	1.13
Y=8	1.78	1.25	1.19	1.18	1.4	1.46
Y=16	1.61	1.2	1.21	1.44	1.45	1.58
Y=32	1.56	1.3	1.49	1.48	1.5	1.62
Y=64	1.54	1.5	1.5	1.5	1.6	

TABLE IX
RUNTIME OF PROGRAM FOR GRID SIZE 128*128 WITH DIFFERENT WORK GROUP SIZES

X. COMPARISON WITH OPENMP AND OPTIMISED SERIAL

Table X lists the fastest runtime of Lattice Boltzmann achieved by different implementations. It can be seen

that OpenCL clearly outperforms OpenMP and serial. It can also be observed that the improvement in performance is more significant on larger grids. While grid size 128*128 experienced a 1.8x performance boost, grid 256*256 sped up by 3.7x and grid 1024*1024 sped up by 5.7x. We can assume the larger the grid, the more advantage the GPU will have. This is because CPU focuses on latency and GPU focuses on bandwidth. GPU is also better at performing large scale fine-grained parallelism, using a large number of threads. Therefore, in comparison, CPU is rather inefficient for this purpose. According to the ballpark timings, a well optimised implementation of Lattice Boltzmann should be comparable with a simple OpenCL implementation. However with further optimisation, it is still very likely that OpenCL can outperform OpenMP in similar applications.

Size	128*128	256*256	1024*1024
Serial (s)	12.2	49.1	176.8
OpenMP (s)	1.5	6.7	27.9
OpenCL (s)	0.8	1.8	4.9

TABLE X
RUNTIME OF LBM IN SERIAL, OPENMP AND OPENCL

Size	128*128	256*256	1024*1024
OpenMP (times)	8.1	7.3	6.3
OpenCL (times)	15.2	27.3	36.1

TABLE XI
RUNTIME SPEED-UP OF LBM IN OPENMP AND OPENCL COMPARED TO SERIAL

XI. PERFORMANCE ANALYSIS

The following equation is used to calculate the maximum bandwidth achieved by this program.

$$\frac{sizeof(float) * nx * ny * NSPEEDS * 2 * maxIters}{runtime} \quad (1)$$

Assuming an input size of 1024*1024 and max iteration of 20000 rounds, the code is able to achieve a memory bandwidth of 286.98 GB/s, significantly higher than the memory bandwidth achieved in OpenMP of 50.4 GB/s. This further confirms the advantage GPU has over CPU when performing mass-scale parallel computing tasks.

The operational intensity of the code is computed by the following equation.

$$\frac{number\ of\ float\ point\ operations}{number\ of\ memory\ access\ opertions} \quad (2)$$

The operational intensity for this program is calculated to be 1.69. And The overall performance can be calculated by multiplying the operational intensity by the memory bandwidth.

$$Bandwidth{*}OI = 286.98{*}1.69 = 454.57 GFLOPS/s \quad (3)$$

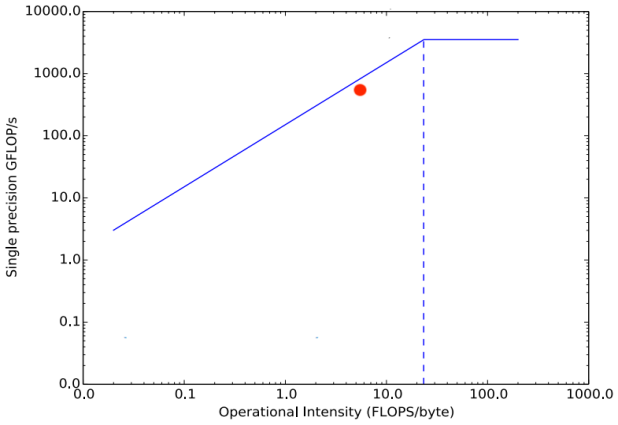

Fig. 1. Roofline model of NVIDIA Tesla P100-PCIE-16GB