High Performance Computing - OpenCL

Lin Feng
Computer Science
University of Bristol
Candidate: 97302
rg18215@bristol.ac.uk

***Abstract*—The architecture of high-performance computing systems is becoming more and more heterogeneous as Heavily Parallelised Compute has the ability to run on both CPU and GPU devices. In this paper, the Lattice Boltzmann Method (D2Q9) code implemented in OpenCL is used and its performance is analyzed on every stage. This is also compared with results obtained with code optimized and implemented in OpenMP. The result shows that specific arithmetic with large data sets using GPU hardware is faster than the CPU.**

***Index Terms*—High Performance Computing; Lattice Boltzmann Method (LBM); OpenCL; OpenMP; Parallel programming**

I. Introduction

Both CPU and GPU have multiple cores in general. However, GPU devices typically have a significantly larger number of cores than CPU. Therefore, GPU ordinarily has extremely high memory bandwidths and it is ideally suited for computation that can run in parallel. Generally, this involves arithmetic on large data sets where the same operation can be performed across thousands of elements at the same time. In addition to this, an efficient computation will need both CPU and GPU to run properly.

The Lattice Boltzmann Method (LBM) is a technique for simulating the movement of complex fluid systems. Fluid systems are used in many industries to transmit signals and power using a network of tanks, pipes, valves, pumps and other flow devices. This paper describes how to accelerate the LBM technique by using OpenCL – the open standard for parallel programming.

Open Computing Language (OpenCL) is a framework for running code on two sets of computer hardware: a host system that relies on one or more CPUs and a device system with one or more OpenCL-enabled GPUs. The part running on the host usually performs initialization and controls kernels running on devices, whereas devices run those parts of the algorithm with thousands of lightweight threads in parallel. To use OpenCL effectively, it is important to understand the differences in the design between the device and the host system and how to determine the performance of OpenCL applications. This performance depends on several factors, such as the instruction set of the architecture or the precision of computations and the number of cells. For this reason, rather than discussing hardware components, it is important to measure the performance of the implemented code in order to best structure the OpenCL code for faster execution.

II. Code implementation

In order to call a kernel that can be executed on an OpenCL device, host side needs to create a context and queue, create and build the program, setup memory objects, define the kernel, pass the data and enqueue commands. The host can read the data from the device after all of this. On the device side, OpenCL programs generate many work-items to perform the kernel functions parallelly, usually each work-item processes one item of the data-set.

OpenCL compiles the data and tasks in the kernel functions. Kernels are executed by a set of work-item and the kernel code specifies the instructions executed by each work-item. Work-items are collected into work-groups and each work-group executes on a computer unit. Each work-item has a global ID and a local ID. The global ID is a unique identifier among all work-items of a kernel. On the other hand, the local ID identifies a work-item within a work-group. Work-items in different work-groups may have the same local ID, but they do not have the same global ID. Moreover, each work-group has a unique group ID.

The OpenCL memory model contains several spaces with different size and access time. The data in global/constant memory is mapped on the entire device; the data in local memory is shared by work-items only

within a work-group, and the data in private memory is used only by individual work-item. When the host transfers data to and from the device, data is stored in and read from global memory. Global memory is the largest memory region on an OpenCL device but it is the slowest for work-items to access. Work-items access local memory much faster than global memory. Local memory is not as large as global memory but, because of its access speed, it is a good place for work-items to store their intermediate results. Finally, each work-item has exclusive access to its private memory, which is the fastest storage place, but it is commonly much smaller than all other address spaces. Therefore, it is important not to overuse private memory. Furthermore, since each work-item and work-group cannot communicate with other work-items and work-groups, it is also important to consider the parallelism instructions.

A. Kernel functions

Firstly, rebound and collision functions were moved into the kernel by following the structures of OpenCL programming.

Comparing the running time of each function in different processors: CPU implementation, CPU parallel implementation OpenMP and GPU implementation OpenCL. The results show that, for small problem-scale and the simple calculation (e.g. accelerate_flow function), CUP incredibly performs the best. It can be considered as CPU cores are typically faster than GPU cores. However, with "if" statements, OpenMP does not perform as well as expected. In contrast, when it comes to the heterogeneous computing (e.g. collision), OpenCL and OpenMP perform almost 10 times better than the CPU implementation. In addition, the propagate stage needs to implement some out of order memory accesses in order to swap data between adjacent lattice points. The rebound stage also needs to read from and write to memory. All of those memory accesses ultimately limit the performance of the code.

	Original	OpenMP	OpenCL
accelerate_flow	0.006 s	1.0+-0.3 s	0.6 s
propagate	4.3 s	1.0+-0.3 s	1.4 s
rebound	0.6 s	1.0+-0.3 s	0.8 s
collision	14 s	2.0+-0.3 s	1.9 s

Overall OpenCL stably performs the best. In OpenCL, data values must be transferred from the host to the device. However, the bandwidth between the device memory and the GPU is much higher than the bandwidth between host memory and device memory. Therefore, these transfers are costly in terms of performance and thus they should be minimized. Programs that run multiple kernels on the same data should favour leaving the data on the device between kernel calls rather than transferring intermediate results to the host and then sending them back to the device for subsequent calculations.

In conclusion, for the best overall application performance, all lattice Boltzmann code should be handled in the kernel by OpenCL in order to keep data on the device as long as possible. Therefore, for this project, all of the functions were moved into the kernel, even there are some stages running kernels on the GPU does not perform any speed-up compared with running them on the host CPU.

B. Optimizations

Secondly, after moving all functions into the kernel, some CPU code optimizations were implemented: merging all stages into only one function; unlooping the small loop to accelerate the calculation of local density, and using a ternary operator to select store value instead of using "if" statement. All of these optimizations aim to reduce memory access and achieve better performance.

Coalesced memory access or memory coalescing refers to combining multiple memory accesses into a single transaction. On the K20 GPUs, every successive memory can be accessed by a warp in a single transaction. However, when memory is not sequential, memory access is sparse or misaligned memory access, these factors may result in uncoalesced load. How the matrix is laid out will affect the memory access.

Thirdly, in order to keep matrix to reside linearly in memory as much as possible, the data type of cells and tmp_cells , t_speed was split into several independent arrays. In other words, using arrays instead of the structure of arrays.

The results show that performance is significantly improved after moving all functions into the kernel since there are no more cells and tmp_cells transfers between the host and the device. The performance becomes even better with optimizations: merging all stages together greatly reduces global memory accesses. Changing the data type slightly speeds up the computation, but every

thread still requires data from the neighbouring lattice points which effects the speed.

	kernel	optimizations	AoS data type
128*128	8.7 s	3.5 s	2.9 s
256*256	68.7 s	19.4s	16.3 s

C. Reduction

At the end of the av_velocity stage, the host still needs to read the data from the device and performs the sum which takes, approximately, 1.5s in total. In order to improve its performance, some different methods to apply the reduction for calculating the av_vels were tested.

The first method is using OpenMP on host performs partial in each thread to do the reduction of an array of elements to a single value. It is the simplest way to improve the performance on the CPU since it only needs one line to apply the OpenMP. However, the performance gets worse to some extent. Considering opening parallel regions is expensive. It is not worth it for simple loops and small data sets, and the data transfer from device to host still cost high memory bandwidth after all.

data sets	running time
128*128	3.8 s
256*256	42.0 s
1024*1024	42.0 s

The second approach is using one work-item to perform a sum within a work-group and store the sum in global memory, then return the array of the partial sum to host. The work-items read from global memory serially which is very inefficient, and it is not a parallel process when only one work-item forms the partial sum.

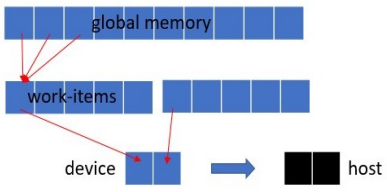

Fig. 1. the second approach

The third method consists of changing work-items to perform the partial sum and copies it to the local memory in parallel, wait for work-items within the same work-group finishing the task and use one work-item to perform the partial sum. Using local memory instead of global memory speeds up the memory access and the size of data transfer is reduced, which leads to better performance.

In this case, the size of the local memory is the critical element for performance. However, tuning the kernel work-group size for GPUs is a challenging problem. There are many factors that may affect the choice of a good work-group size. It depends on the data re-use ratio and coalescedness of access pattern. Random access to local memory is much faster than random access to global memory. While using too much local memory/private file, it can be even slower because more local memory consumption leads to less occupation, less memory latency hiding, and more register spilling to global memory.

Therefore, in order to find the best fit work-group size, their running time was compared by applying different size of local memory: 88, 1616 and 3232. According to the results, obviously using 1616 as the local memory size achieves the best performance. However, it is still not an efficient way of using only one work-item to perform the partial sum.

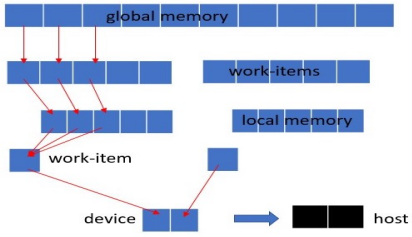

Fig. 2. the third approach

	8*8	16*16	32*32
128*128	2.6 s	2.4 s	5.9 s
256*256	11.5 s	12.0 s	27.0 s
1024*1024	32.6 s	24.3 s	94.0 s

The fourth technique is using binary addition to improve on the linear sum. Same as the last solution, work-items perform the partial sum and copy it to the local memory in parallel but, this time, adding one-half values to the other repeatedly until getting the global sum.

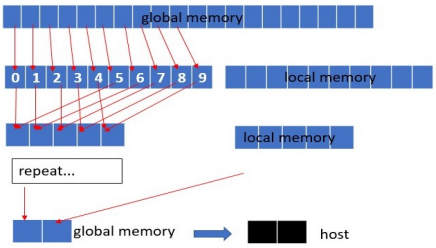

Fig. 3. the fourth approach

By forming the sum parallelly in local memory, the performance gets plainly better compared to the former solution. However, reading the data from the device and performing the sum on the host still takes 0.5s roughly for 128128 data set.

data sets	running time
128*128	1.9 s
256*256	8.2 s
1024*1024	19.1 s

Finally, setting the global work size to be half of the input array length and slightly re-order to remove a couple of loop iterations. Then, perform the sum parallelly as in the former solution. In order to change the global work size without affecting other stages, the av_velocity kernel was separated. Use 16, 16*16, 32*32 respectively as local work size to implement the reduction.

However, the running time is about from 2.5s to 2.8s and it does not improve the performance. There are some factors affecting that can be considered. In this case, each work-item needs to do a double calculation task including the arithmetic of local density and so on. Therefore, this can cost more time than expected when it only adds up two of the input values.

III. Conclution

OpenCL Lattice Boltzmann implementations work by creating hundreds of threads, all working in parallel. Combine all stages in the OpenCL kernel gets the better performance than CPU and OpenMP implementations. In addition, doing reduction parallelly before reading the data from the device improves the performance even more. The best running time I achieved shows below.

However, the propagate stage requires data from the neighbouring lattice points which still cost 1s more or less, global memory access ultimately limits the available bandwidth. Could use tiles hola exchange strategy, optimize delivery of the data by doing a cached copy, creates more than one OpenCL pipelined kernels, where more pipelines can do more floating logic performed in parallel.

data sets	running time
128*128	1.9 s
256*256	8.2 s
1024*1024	19.1 s

References

[1] Wiki.tuflow.com. (2019). Hardware Benchmarking Topic HPC on CPU vs GPU Tuflow. [online] Available at: https://wiki.tuflow.com/index.php?title=Hardware_Benchmarking_Topic_HPC [Accessed 4 Apr. 2019].

[2] A Portable OpenCL Lattice Boltzmann Code for Multi- and Many-core Processor Architectures. [online] Available from: https://www.researchgate.net/publication/262932546_A_Portable_OpenCL_L_and_Many-core_Processor_Architectures [Accessed 4 Apr. 2019].

[3] Nallatech. (2019). FPGA Acceleration of Lattice Boltzmann using OpenCL. [online] Available at: https://www.nallatech.com/fpga-acceleration-of-lattice-boltzmann-using-opencl/ [Accessed 4 Apr. 2019].

[4] Cs.cmu.edu. (2019). [online] Available at: https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/OpenCL_Best_Practices_Guide.pdf [Accessed 4 Apr. 2019].

[5] Ukopenmpusers.co.uk. (2019). [online] Available at: https://ukopenmpusers.co.uk/wp-content/uploads/uk-openmp-users-2018-OpenMP45Tutorial_new.pdf [Accessed 4 Apr. 2019].

[6] Cl.cam.ac.uk. (2019). [online] Available at: https://www.cl.cam.ac.uk/teaching/1617/AdvGraph/07_OpenCL.pdf [Accessed 4 Apr. 2019].

[7] Fz-juelich.de. (2019). [online] Available at: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-05-reduction.pdf?__blob=publicationFile [Accessed 4 Apr. 2019].

[8] Web.engr.oregonstate.edu. (2019). [online] Available at: http://web.engr.oregonstate.edu/ mjb/cs575/Handouts/opencl.reduction.2pp.pd [Accessed 4 Apr. 2019].

[9] Dournac.org. (2019). Parallel Sum Reduction GPU/OpenCL versus CPU. [online] Available at: https://dournac.org/info/gpu_sum_reduction [Accessed 4 Apr. 2019].

[10] Memory bandwidth. [online] Available at: https://fgiesen.wordpress.com/2017/04/11/memory-bandwidth/ [Accessed 4 Apr. 2019].

[11] Layton, J. (2019). Benchmarking Memory Bandwidth ADMIN Magazine. [online] ADMIN Magazine. Available at: http://www.admin-magazine.com/HPC/Articles/Finding-Memory-Bottlenecks-with-Stream [Accessed 4 Apr. 2019].

[12] Cvw.cac.cornell.edu. (2019). Cornell Virtual Workshop. [online] Available at: https://cvw.cac.cornell.edu/gpu/coalesced [Accessed 4 Apr. 2019].