AHPC Coursework: MPI + X

Xiang He, xh17500

I. Parameters and setup

A. Experiments setup

In this task, every experiment result is obtained from the average of 5 experiments. Run time is recorded with 2 decimal places. While compiling with mpicc, `-std=c99` is added to avoid unexpected errors. And `-O3` and `-ffast-math` flags are added for optimisation.

B. Limiting factor analysis

From the roofline model of BlueCrystalp4, we know when running the serial code in single core, the operation intensity of Lattice Boltzmann code is in the memory bandwidth bound area of roofline model. That may indicate a significant fraction of execution pipeline slots could be stalled due to demand memory load and stores.

II. Serial Optimisation

For each cell in Lattice Boltzmann (LBM), the main tasks are calculating `accelerate_flow`, `propagate`, `rebound` and `collision`. In the end, average velocity is calculated. The for loop of `propagate`, `rebound`, `collision` and `av_velocity` can be fused together to get better performance (a swap of pointers is necessary to fuse `propagate` in). Besides, the calculation of `tot_cells` only needs to be implemented once since it is a consistent value. Thus it is calculated out of the iteration. The calculation of index $ii + jj * params.nx$ within the for loop has been executed for many times. Thus they will only be calculated once after the optimisation. All of the divisions in the collision step are estimated before the for loop since they are all consistent with every cell.

III. MPI Parallelisms

We use MPI to split our work across multiple processors and each smaller work is owned by a different rank. Ideally, each rank only needs to send their works back to master rank at the end of the program. However, for each timestamp in LBM, at the `propagate` stage, each cell needs to get the speeds of its all 8 neighbours to do a propagation. And that is where the Halo Exchange happens.

A. Halo Exchange

Every rank needs to access data held on a different rank. But if we send all the data they require before every `propagate` step, the job is done. First of all, the problem is how to decompose the whole grid. If we decompose it by columns, since the address of data we need to send is not consistent, a procedure to collect the data is needed. If we decompose as tiles, many calculations for the start and end position of every sub-grid is required. Thus, a decomposition by rows is selected here. Another problem is that there may be remaining rows if we split the cells into every rank. Adding all the remaining rows to the last rank is the easiest choice but it will cause load imbalance and the performance cannot be scaled very well with the increase of the number of ranks. Thus, **the remaining rows are divided again to several single rows and will be added to every rank** (e.g. remain 4 rows, and the rank from 0 to 3 will add one more row respectively) to make a good load balance (see Figure 1).

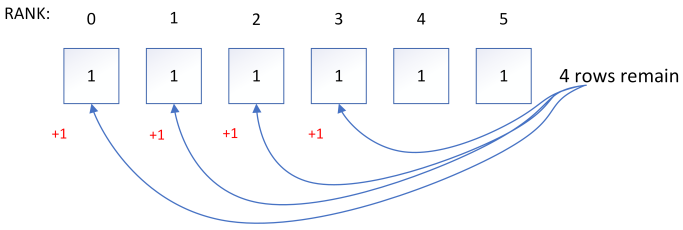

Fig. 1: Load Balance

And then the communication pattern should be selected to make a Halo Exchange. A send-receive operation is beneficial for conducting a communication operation across a number of processes or a number of nodes. If a blocking communication pattern are used for such a communication, then the correct order of sends and receives is very important (for instance, even ranks send, then receive, and odd ranks receive, then send) in order to prevent cyclic dependencies on each other which can lead to a deadlock. `MPI_Send` and `MPI_Recv` are both blocking operations, which means that the call will not return until the function arguments are safe for re-use. It is safe for communication, but the ranks will be idle waiting for the corresponding receive operation to be posted to make a pair. Obviously, it is vulnerable to deadlock. To avoid deadlock, A good communication pattern is every rank send to its left and right neighbours and receive from them simultaneously with a single `MPI_Sendrecv` function. With this function, the system will take care of the order problem automatically. However, The semantics of a Sendrecv operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads. Therefore, `MPI_Sendrecv` suffers from exactly the same "`MPI_Send` may block" issue as regular standard mode sends. The counterpart one is `MPI_Isend`, which is a non-blocking call so will return immediately and will never cause deadlock. A `MPI_Wait` call via additional request is needed to

make sure the completion of communication since the function arguments will not be safe for re-use until the communication is complete. A call to `MPI_Waitall` returns when the operation identified by request is complete. While the ordering of the individual calls to `MPI_Isend` and `MPI_Irecv` do not matter for correctness (because theyre all progressed in `MPI_Waitall`, regardless of the initial posting order), the ordering may matter for efficiency. Some other works can be done between the communication calls and wait for function to avoid idle processes and thus to be more efficient. Since I only put the `accelerate_flow` step between the Halo Exchange and the wait function, the improvement is limited. An ideal implementation is to calculate all the rows except the first and the last one for every rank between communication calls and the `MPI_Waitall` function. And then calculate the remaining cells. Since time is limited, this implementation in not provided.

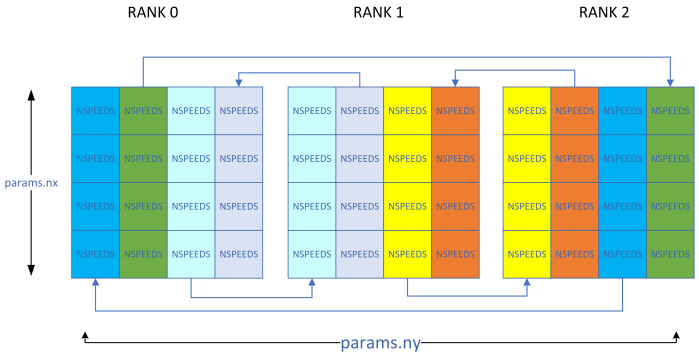

Fig. 2: Halo Exchange

B. Reduce and Gather

The `av_velocity` variable is the average velocity of every non-obstacle cell. Since this calculation is executed within the whole grid, the local velocities need to be sent back to the master rank. A typical implementation is to use `MPI_Recv` in master rank to receive the local velocities and use `MPI_Send` in other ranks to send to local velocities. A more straightforward and faster solution is to apply an MPI built-in function: `MPI_Reduce`, which can be 3s faster than the first one in my test. The final results are reduced to the master rank.

After all the iterations, local cells need to be delivered back to master rank to write the final values. This procedure can be implemented by `MPI_Gather`, which takes elements from many processes and gathers them into one single process.

C. Performance Analysis

The performance speedup for every input file is given in Figure 3. We can tell that only the $1024 * 1024$ one can scale very well among multi-cores. This is mainly because for other input files, the amount of task is too small that the sub-task for every rank will not be affected too much when there are enough cores. Also, the Halo Exchange will take some time to deal with. And the remaining rows will also affect the scalability. Thus, the performance cannot be scaled well ideally. But still, the performance has been improved significantly with MPI parallelism. It is worth to mention that the scale on single node is worse than scale on more nodes since the memory bandwidth can be bigger.

TABLE I: Comparison between blocking and non-blocking calls (1024 * 1024)

	MPI_Sendrecv	MPI_Isend & MPI_Irecv
Time	8.77s	8.13s

TABLE II: Final Results of Flat MPI

	128*128	128*256	256*256	1024*1024
serial	22.83s	48.00s	201.03s	808.47s
MPI	0.61s	0.88s	3.09s	8.13s

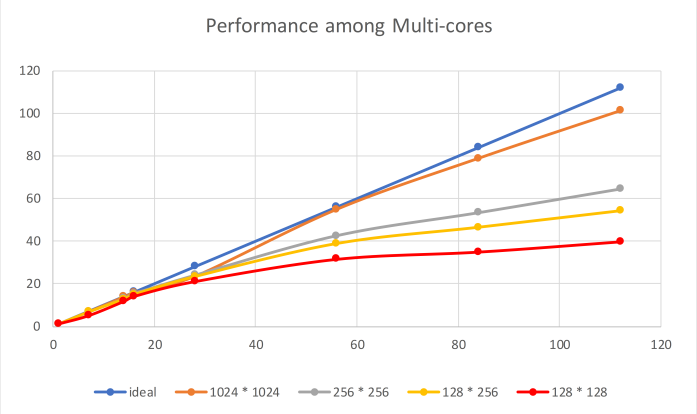

Fig. 3: Scaling performance

IV. OpenCL

OpenCL is selected for GPU implementation of LBM. GPU is throughput optimised, or latency tolerant, which means the optimisation focuses on throughput: memory throughput and instruction throughput. Memory throughput can be improved by running as many operations as possible at once to cache the memory throughput. And the instruction throughput can be optimised by a proper memory access scheme.

A. Memoery Access & Data Structure

OpenCL has a unique memory model. The global memory region contains global buffers and is the primary conduit for data transfers from the host (Intel Xeon E5-2680 v4) to/from the OpenCL devices (Nvidia Tesla P100-16GB). This region will also contain OpenCL C program code that will

be executed on the OpenCL devices, but the access to this region has long latency. The local memory region, where the latency is 100x smaller than global memory, is not defined to be accessible from the host. This memory is shared within a work-group. It can be viewed as a core local scratchpad memory. The use case for local memory is for an OpenCL work-group to migrate a portion of a global buffer to/from a local buffer for performance improvement since the access to data in local memory will be much faster. This use case is optional for users as access to global buffers in DDR will be cached in both the L2 cache and the L1D cache. However, performance can often be improved by taking the extra step in OpenCL C programs to manage local memory as a scratchpad. The top one is the private memory, which is owned by every work-item and cannot be shared. If we want the algorithm to run really fast, we need to split the grid of cells into sub-grids that are small enough to fit into the fastest level of the memory hierarchy. For LBM, local memory is only utilised for reduction of average velocity.

Global memory latency is usually around 400 - 800 circles. Optimisation for global memory access is crucial. Coalescence memory access means if thread i accesses memory address n, then thread $i+1$ should access memory address $n+1$. It is the key for high bandwidth. The reason is GPU device can read 32-bit, 64-bit or 128-bit words from global memory into registers in a single instruction. And data structures and arrays should align on these byte boundaries to minimise the number of instructions since accessing global memory is quite costly. So global memory bandwidth can be more efficient when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can coalesce into a single memory transaction of 32, 64 or 128 bytes. Otherwise, a separate memory transaction is issued for each thread and throughput will be significantly reduced. Misaligned access and Strided access are both time-consuming.

Since Array of Structures (AoS) suits cache hierarchies on CPUs well and does not suit memory coalescence on GPUs, Structure of Arrays (SoA) is a better choice for GPU implementation, where adjacent work-items can access adjacent memory. It means that the structure of cells should be changed. The implementation selected is removing the speeds structure and put all the same direction of speeds together (first line stores the first speed of all cells). And the performance comparison is provided in Table III.

TABLE III: Comparison between SoA and AoS

	SoA	AoS
Time	4.21s	5.87s

B. Branching

GPU tends not to support speculative execution, which means branch instructions will have high latency. When different work-items are executing within a same SIMD ALU array take different paths through conditional control flow, it will cause divergent branches. And work-items will be idle when waiting for other work-items to complete their works. Thus the performance will be reduced. An ideal solution is to convert the conditional control flow to straight line code. Here we transform the conditional control into selection and masking to avoid branches.

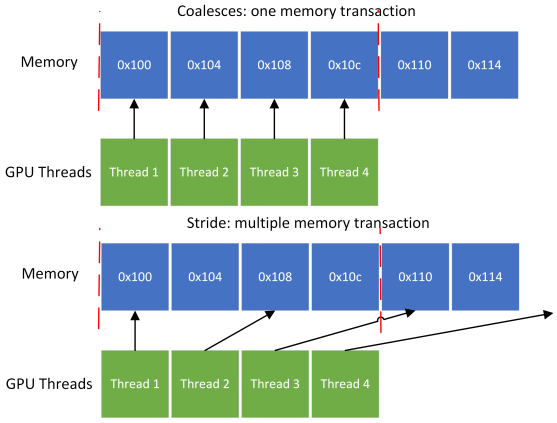

Fig. 4: Coalescence Memory Access

C. Reduction

Reduction in kernel function is different from the CPU version. Assume there are 8 work-items in a work-group, then the reduction scheme will be like what Figure 5 shows since you cannot use a for loop to add them to the first work-item directly. `barrier(CLK_LOCAL_MEM_FENCE)` is needed to synchronise all work-items within a work-group. Every work-item will wait until all work-items in its work-group reach the barrier. This is a reduction within a work-group. However, there is no barrier-like synchronisation between two different work-groups. Therefore, the final sum by adding up all the partial sums of every work-group will be carried out in the host function. For LBM, this reduction is executed for the calculation of `av_velocity`. The local velocity of every work-item is stored in the local memory which can be shared between them, so they do not need to access global memory while processing the reduction procedure.

D. Kernel Functions

Two kernel functions are used in this implementation: one is for `accelerate_flow`, the other is for remaining four steps. There will no explicit for loop in kernel functions if the indexes are got from `get_global_id()`. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. Valid values of index are from 0 to `get_work_dim()` - 1. It is worth to mention that the reduction of average velocity is performed within a work-group. Thus the local ID can be obtained from `get_local_id`. The local size is a key optimisation for GPU (in this task, only the reduction part is

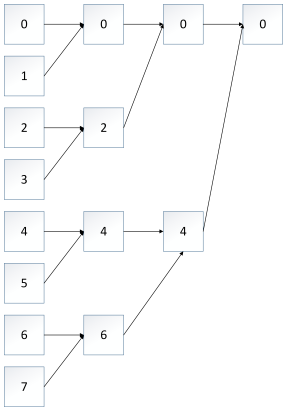

Fig. 5: Reduction Procedure

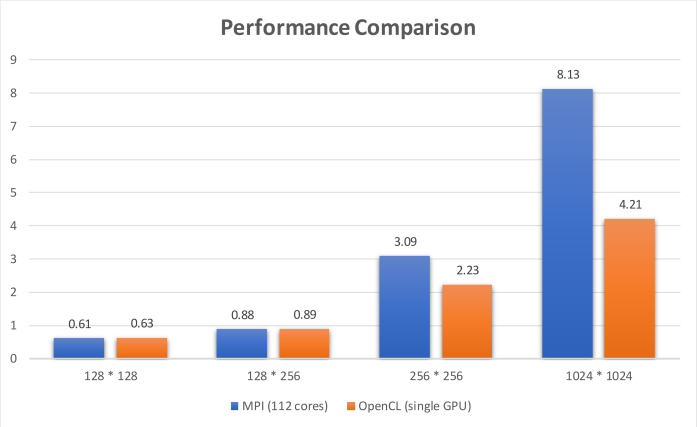

Fig. 6: Reduction Procedure

relevant). If it is too large, the register will spill to memory and cause cache misses. And if it is too small, the local memory cannot be utilised efficiently, and the latency will not be hidden. There is no rule of thumb here. In my experiment, the best performance is set to {16, 16}. And in the second kernel function, the temporary values of `cells` are stored in a private array `tmp` since it will be used for many times after the `propagate`. It does not need to access the global memory for these values anymore, and private memory is much faster, so the performance can be improved a lot.

Besides, in the kernel functions, `#pragma unroll` is used for loop in the 9 speeds. The principal goal of the `#pragma unroll` is to improve the performance of loops via unrolling. Typically this enables other optimisation or enhances instruction level parallelism of a thread. This pragma is supported only when option `-O3` is set.

E. Performance Comparison

The performance comparison of flat MPI version and single GPU version is shown in Figure 6. It is obvious that the performance of OpenCL on single GPU is greatly faster than 112 cores version of MPI implementation.

V. MPI + OpenCL

If we want the program to run on more than one GPU, the communication between these GPUs can be performed by MPI. To run it on 4 GPUs, two nodes are required, and every node has 2 MPI ranks, and every rank run an OpenCL task. MPI holds the Halo Exchange between the ranks and writes the corresponding arguments into the global memory of OpenCL devices. In each GPU, the partial calculation is performed within kernel functions. And after that, the value of `cells` will be read back into host memory to do the Halo Exchange. After the main iteration, the cells which have been read back from GPUs will be sent back to master rank for every MPI rank. And the reduction for `av_vels` is performed within every rank (among work-groups) first; then the final results are reduced among every rank (there are three stages for the reduction).

Besides, since the cells need to be read back from GPUs to perform Halo Exchange and write into GPUs again for each iteration, `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` will be very time-consuming because they require the contents of the buffer to be copied. In this case, `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject` can be faster for data mapping. The map/unmap functions map the underlying memory store of a buffer into the host address space and allow the host application to read and write from/to the buffers content directly. It means that if we use `CL_MEM_READ` and `CL_MEM_WRITE` flags simultaneously while mapping the buffer, every change to the return pointer will be written into the buffer and vice versa. For better performance, every iteration will only map (read and then write) the corresponding two rows of `cells` for communication. Since it is still time-consuming for mapping data (for instance, half of the elapsed time is mapping data when using 1024 * 1024 input) for each iteration and Halo Exchange also needs some time, the MPI + OpenCL version will be slower than flat OpenCL on single GPU, and the scalability is awful although the calculation itself is faster with multi-GPUs.

TABLE IV: Performance among multi-GPUs

GPU number	1	2	4
128 * 128	2.79s	3.08s	3.29s
256 * 256	7.90s	6.98s	7.04s
1024 * 1024	9.74s	6.11s	4.45s

VI. Conclusion

MPI is a powerful tool for communication between different processes and different nodes. And OpenCL is an open, royalty-free standard for cross-platform, parallel programming of diverse processors. The performance of hybrid of OpenCL and MPI can be improved in the future.