ASSIGNMENT 2: MPI+X PARALLELISM

Shuai Ke - 1609628

May 4, 2018

1 Introduction

The lattice Boltzmann code is improved in four ways in this assignment, there are Serial Optimizations, Flat MPI, OpenCL, and MPI+OpenCL. The report contains the descriptions of the implementation, the comparisons about MPI+OpenCL vs. Flat MPI (CPU) and CPU vs. GPU (MPI+OpenCL), and an analysis of the results. The *Figure 01* shows the summary of the best run-times in each section, comparing with original run-times. The Y-axis in the left is the run-times in data size is 128x128 and the Y-axis in the right is in data size is 1024x1024.

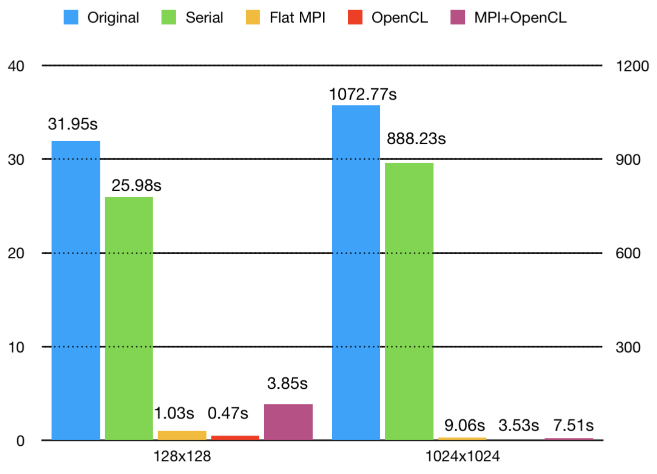

Figure 1: The Summary of the best Run-times

2 Serial Optimizations

The optimizations in this section are not used MPI or OpenCL, there are some serial optimizations, like reduction of computation, memory layout. It also adds some compiler flag, such as *-ffast-math* and *-msse* to vectorize the program.

2.1 Merge Functions & Computations

Before parallel the program, there are some serial optimizations, including loop fusion, reduction of calculation and replacing division with multiplication. The loop fusion merges the rebound, collision and av_velocity operations, which means generating the results in a loop over all cells instead of three times. Furthermore, the original code will calculate the index every time during it accesses the array of cells. To reduce the extra computational cost, the optimization calculates the index in advance and stores it in a variable which reduces around 80x1024x1024 computations in each round when the data size is 1024x1024. Since the division operation is complicated, the serial optimization also replaces the division with multiplication operation. The *Table 1* shows the performances of serial optimization and origin.

data size	128x128	128x256	256x256	1024x1024
original	31.95s	64.49s	257.66s	1072.77s
serial	25.98s	52.48s	212.32s	888.23s

Table 1: Performance of Serial Optimizations

2.2 Memory Layouts

Besides, there is an attempt at memory layout, it changes the data format from the array of structure (AoS) to the structure of arrays (SoA). The definition of the SoA looks like *float cells[NSPEEDS][ny][nx]*. It does not improve the performance since thrashing cache. The program accesses cells[0][y][x] to cells[8][y][x] in sequence, but their memory addresses are discontinue. It will lead to many data requests miss the cache because the system usually loads the data which close to the previous access address. Therefore, the program will keep using the array of structures. The run-times comparisons with different data size show in *Table 2*.

data size	128x128	128x256	256x256	1024x1024
AoS	31.95s	64.49s	257.66s	1072.77s
SoA	160.32s	60.57s	612.11s	2500.12s

Table 2: Performance with Different Memory Layout

3 Flat MPI

By using the MPI, the program can be paralleled in multiple cores. This section introduces some optimizations with the MPI and compares different strategies of the implementation. Halo exchange by row and column will be compared firstly; then the load balance issue is discussed; in the third part, it describes a method for summation of *av_vels* in multiple processes; after that, it is an investigation about different methods of data exchange; the scale analysis is in the end.

3.1 Halo Exchange by Row and Column

Halo exchange by row means decomposing the grid of *cells* by row and each worker computes the range of rows. After each round, the adjacent processors will change the data with each other by using *Sendrecv* API, and all processors send the local value of *av_vels* to MASTER by *Send* API after

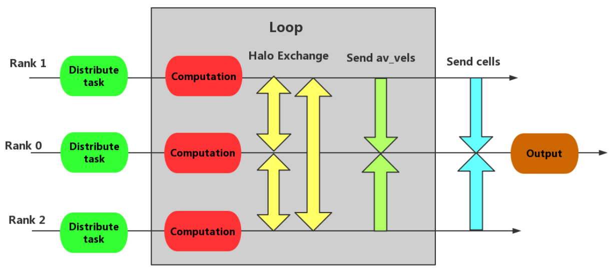

Figure 2: The Schematic Diagram of Halo Exchange by Row

calculating. The *Figure 2* is the schematic diagram of the program.

The halo exchange column is similar to by row, but it decomposes the grid by column. Both of them will distribute the rest works to the last processor when the row or column cannot be divided averagely. However, halo exchange by row has some extra operations like copy the data from *cells* to the buffer before exchange data and from the buffer to *cells* after exchange data since the memory addresses of a column are discontinued. Therefore, the performance of halo exchange by row is slightly better than by column. The comparison of halo exchange by row and column with 256x256 data size is shown in *Table 3*.

processors	1x16	2x16	3x16	4x16
column	15.13s	11.04s	20.89s	9.27s
row	15.03s	9.86s	19.83s	8.15s

Table 3: Comparisons of Halo Exchange by Row and Column

3.2 Load Balance

The performance of 3x16 processors is worst then 2x16 processors in *Table-3* since the load imbalance. The last processor in 3x16 processors will be distributed 21 rows, while it only 8 rows in 2x16 processors. To load balance, the program distributes the rest of task averagely. For example, there are ten rows and four processors. After the first round of allocation, each processor has two rows and two rows remain. The second round will allocate the reaming rows to the first and second processors instead of leaving to the last worker. The *Figure 3* shows the relation between the performance and the maximum number of tasks in the non-balanced and balanced program when the data size is 128x128. It clears that the lines are matching since the performance depends on the slowest process which has the most number of tasks.

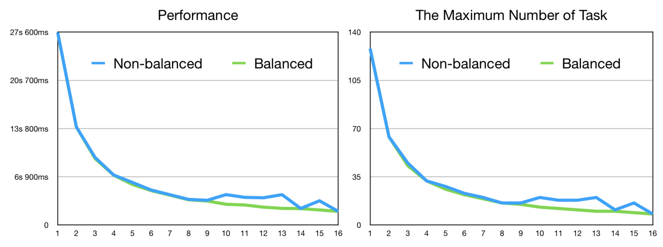

Figure 3: The Relation between Performance and Maximum Number of Tasks

3.3 Reduction

All processors need to send the local *av_vels* to the MASTER processor by using *Send* API during the computation of *av_vels*. Since it is a simple summation calculation, this optimization uses MPI reduction to improve it. The *MPI_Reduce* API takes an array of input elements on each process and returns an array of output elements to the root process. It reduces the complexity of the *av_vels* computation from $o(n)$ to $o(logn)$ since it distributes the works to all processors instead of doing in one processor. For instance, there are 8 processors, it needs three units rather than seven since two units doing by processor 4 and processor 2 and 6 each do a unit. (*Figure 4*).

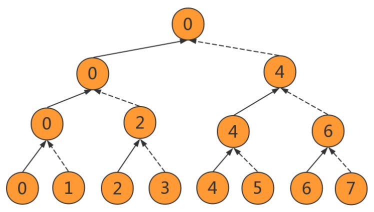

Figure 4: The Example of MPI Reduction

The performance comparisons about reduction and without reduction are shown in *Table 4* (processors 4x28).

processors	128x128	128xx256	256x256	1024x1024
non	3.03s	3.68s	10.4s	19.7s
reduction	1.03s	1.53s	5.26s	9.06s

Table 4: Comparisons of Halo Exchange by Row and Column

3.4 Communication Patterns

This part will describe two extra communication patterns that I have tried, there are *MPI_Send* and *MPI_Isend*. In the beginning, the program only uses the *MPI_Send* API to exchange the data with different processors. This function is asynchronous when there is a system buffer, but blocking if copying large messages into a buffer since its own overheads. The second one is *MPI_Isend*. It is a non-blocking API, but the program needs that the data is exchange completely before the next round. Hence, the *MPI_Wait* is used to wait for full data exchange. There is no significant improvement in performance since it not has any time-consuming operation between *MPI_Irecv* and *MPI_Wait*. The *Table 5* shows the performances of three communication patterns.

data size	128x128	128x256	256x256	1024x1024
Send	1.20s	1.68s	4.94s	9.73s
Isend	1.12s	1.50s	4.90s	9.38s
Sendrecv	1.03s	1.53s	4.85s	9.06s

Table 5: Performances of Three Communication Patterns

3.5 Scale Analysis

According to the **Amdahl's Law**, we can calculate the ideal parallel speed-up $(\frac{1}{P/N+S})$ and the real parallel speed-up $(\frac{T_1}{T_p})$. Notice that the parallelizable fraction is 100% since we only time the parallel part. The line chart (*Figure 5*) shows the comparison, the X-coordinate is the processor number and the Y-coordinate is the value of speedup (the real processor number is 4 x the processor number in the right figure).

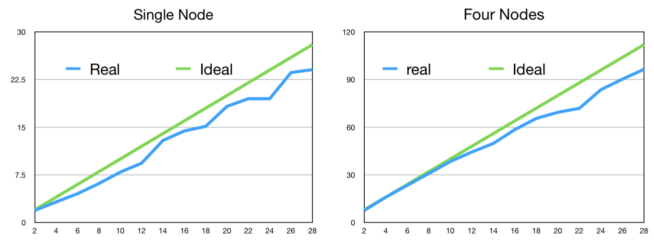

Figure 5: The Comparisons of Speed-up in Real and Ideal

The real speed-up does not match the ideal speed-up since the program needs changing the data between two processors and the extra cost leads to the unexpected result. Additionally, the real speed-up line is not smooth because some processor needs to do more task when the row cannot be distributed completely averagely like processor number is 18 or 24 in the single node. After a lot of iterations, a tiny extra cost in each round will be enlarged in the final result.

processor	8	16	24
single	6.136	14.414	19.491
four	7.628	15.94	23.405

Table 6: The Comparisons of Speed-up with Same Processors and Different Nodes

There is also a comparison of the same processor number with different nodes (*Table 6*). The results in four nodes are better than single node single since it provides more memory bandwidth bound in four nodes.

4 OpenCL

In order to optimize the program by GPU, I implement the optimization by using OpenCL in single GPU. This part describes the basic implementation firstly, then introduces some optimizations such as OpenCL reduction, using private memory and changing memory layout.

4.1 Implementation

There are three kernel functions in the basic implementation, *accelerate_flow*, *propagate* and the function call *move* which merges the remaining functions (*rebound*, *collision*, *av_velocity*) and it will store the return value of *av_velocity* in the array (*rets*). The CPU will let the GPU do the main computation works and sum the *rets* as the value of *av_vels* after GPU finishs the work in each round. It reads the *cells* array after all iterations instead of reading in each round. The *Figure 6* is the schematic diagram of basic OpenCL program.

4.2 Recuction & Private Memory

Since the basic OpenCL program does not show a good performance, there are some optimizations for improving the

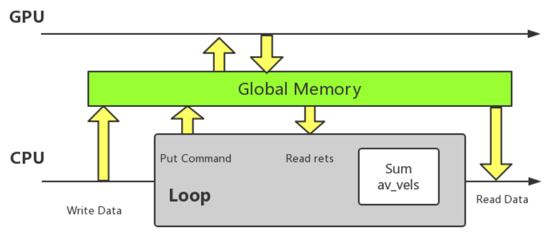

Figure 6: The Schematic Diagram of basic OpenCL Program

run-time. Firstly, the reading and summation of *av_vels* moves out of the loop to avoid frequent data change with global memory. Secondly, using local memory to reduce the computation of *av_vels*. It means that each work-item stores the local *av_vels* in the local memory which is shared with the work items in the same workgroup. The calculation tasks are distributed to different work-items, and the result of the whole workgroup stores in the *rets*, which reduces the calculated amount in CPU. The last optimization is using private memory and merged *propagte* and *move* functions to avoid GPU frequently access the data which in the global memory. It will loads the *cells[j][i]* to the private memory, and write back to global memory after computation. The performances of basic and optimized code are shown in *Table 7*.

processor	128x128	128x256	256x256	1024x1024
basic	3.79s	6.1s	24.72s	223.07s
optimized	0.82s	1.17s	4.11s	13.13s

Table 7: The Performances of Basic and Optimized OpenCL Programs

4.3 Memory Layout

This optimization changes the memory layout from AoS to SoA since the GPU prefers to access memory continuously in successive work-items. However, the implementation in kernel function result in the performances have not improved firstly. In the kernel function, the work-item handles the $jj * nx + ii$ cells. The distributed pattern that *jj=get_global_id(0), ii=get_global_id(1)* leads to the GPU accessing global memory discontinuously, but it can be solved by *jj=get_global_id(1), ii=get_global_id(0)*. While the *Figure 7* shows the schematic diagram of the global memory access.

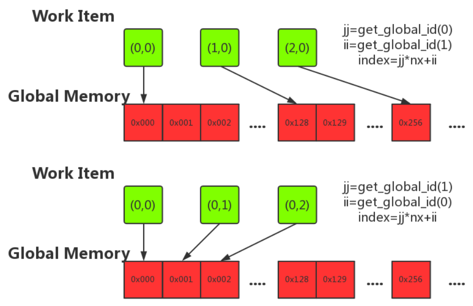

Figure 7: The Schematic Diagram of Halo Exchange by Row

The performance has a huge improvement and the comparisons between AoS and SoA are shown in *Table 8*

processor	128x128	128x256	256x256	1024x1024
AoS	0.82s	1.17s	4.11s	13.13s
SoA	0.47s	0.51s	1.23s	3.53s

Table 8: The Performances of Basic and Optimized OpenCL Programs

Referring to the *Section 2.2*, there is a phenomenon that CPU prefers AoS and GPU prefers SoA in this case since we can arrange GPU to access the continuous global address space in SoA.

5 MPI+OpenCL

In this section, the optimization combines MPI with OpenCL to implement the multiple GPUs program. There are some comparisons between MPI plus OpenCL and Flat MPI, MPI plus OpenCL in CPU and GPU.

5.1 Implementation

For multiple GPUs program, the optimization decomposes the array by rows, and exchanges the data which the neighboring GPU needed between different GPU by *MPI_Sendrecv* API in each round. The program will let the GPU do the computation firstly, then read the specific lines of data and send to neighboring GPU. It also receives the data from neighboring GPU during sending data stage and write it to the global memory for GPU calculation. To facilitate reading and writing data from global memory, the implementation uses AoS. The *Figure 8* shows the procedures of this optimization.

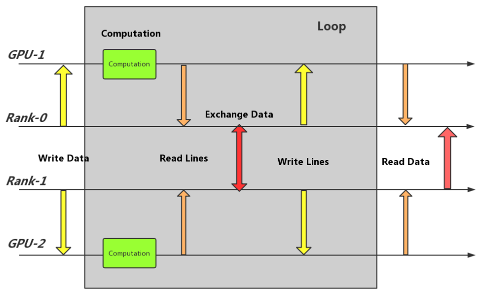

Figure 8: The Schematic Diagram of Multiply GPUs Program

This optimization distributes computation tasks to different GPU, which improves computing power, but it also has an extra cost of changing data between different processors, hence the performance of MPI plus OpenCL is worse than OpenCL in a single node (15.49s vs. 13.13s). The run-times of MPI plus OpenCL with different number of GPUs are shown in *Table 9* when data size is 1024x1024.

nodes	1	2	4
run-time	15.49s	9.36s	7.51s

Table 9: The Performances of MPI plus OpenCL

5.2 MPI plus OpenCL vs. Flat MPI in CPU

In order to compare the performances between MPI plus OpenCL and Flat MPI in CPU, I run the MPI plus OpenCL in CPU only. The performances with data size 1024x1024 are shown in *Table 10*. The results show that the OpenCL can also improve the performance in CPU.

nodes	1	2	4
MPI+OpenCL	92.48s	45.34s	31.64s
Flat MPI	888.23s	464.71s	266.78s

Table 10: The Comparison of MPI plus OpenCL with Flat MPI

5.3 CPU vs. GPU with MPI plus OpenCL

This part compares the performances of MPI plus OpenCL running in CPU and GPU, which indicates that GPU has a better performance in the repeated and simple computation. The GPU is suitable for compute-intensive programs, it has many computing units for parallelization and the global memory for improving data throughput. While the CPU has a better performance in logic control and serial program because it has a high frequency of clock cycles for adapting to the needed for diversity.

nodes	1	2	4
CPU	92.48s	45.34s	31.64s
GPU	15.49s	9.36s	7.51s

Table 11: The Comparison of CPU and GPU

6 Conclusion

In this report, the optimizations are classified into four sections. The first one is serial optimizations which improve the program without additional tools. The second section is based on MPI to optimize the program, it discusses some factors which will affect the run-times, like load balance and reduction. The best performance of 1024x1024 data size is 9.06s and there is a scale analysis from 1 cores up to 112 cores. Then, it is the optimization by using OpenCL with single GPU. Except for the normal optimizations, the memory layout analysis shows that the GPU will have an excellent performance when it can access global memory continually in successive work-items. The final section combines the MPI and OpenCL to implement multiple GPUs program. Although it increases the computational power of the program, the structure of the code is also complicated. Furthermore, there are some performance comparisons, like MPI+OpenCL with Flat OpenCL in CPU, MPI+OpenCL in CPU and GPU.