Implementation and Optimization of Lattice Boltzmann Method for Fluid Flow on GPU with CUDA

Zhangrong Qin, Haiyan Liu, Liuliu Mo, Yuanyuan Li

College of Computer Science and Information Engineering, Guangxi Normal University, Guilin 541004, China, haiyan1l@163.com

Abstract

In recent years, the computational fluid dynamics has been widely used in the aviation, weather, chemical industry and so on. The demand on computing power for modern computational fluid dynamics is getting higher and higher. Aiming at the features of the LBM (lattice Boltzmann method), we put forward a parallel data structure layout of the D2Q9 model and related algorithm based on CUDA (compute unified device architecture). Taking the plane Poiseuille flow as a test example, we implement the parallel computation of the LBM based on LBGK (lattice Bhatnagar-Gross-Krook) model on GPU (graphic processing unit). By comparing the results obtained on a NVIDIA GeForce GTS 450 GPU with those on a Intel Core i5-760 quad-core CPU, both have good agreement and the highest speedup of GPU is about 102 times faster than that of CPU. The result indicates parallel computation of the LBM on GPU is completely feasible, which provides a very effective way to solve the complex problem of the modern computational fluid dynamics on a low cost computer.

Keywords: CUDA Technology, GPU, LBM, Parallel Computing, Multithreading

1. Introduction

In recent years, the lattice Boltzmann method (LBM) has been developed into an alternative computational fluid dynamical method, and successfully applied in a variety of fields, such as complex flows, aviation, weather, chemical industry and so on. With the development of the applications of LBM, the problems are more and more complicated, and the demand for computing power is higher and higher. The computing performance of computer needs computing power of per second 10 billion, 100 billion, even trillion times, the general computer system is usually difficult to do the work, a high cost and high performance computer, even a super computer can meet the need. How to carry out these computing tasks at low cost with high efficiency has become a research focus. GPU (Graphic Processing Unit) is a multi-core processor in 3D graphics card which has the parallel processing structure inside the chip and strong ability of floating point calculation. In 2007, the NVIDIA company launched a CUDA (compute unified device architecture) technology based on general purpose computation on GPU. This framework provides similar C language development environment, designer don’t need to know complex graphics API knowledge and can develop a CUDA program that uses the GPU resource to compute directly by using C language and the CUDA extension library. This can reduce the complexity of development and enhance efficiency. Now, CUDA technology in the parallel computing has been widely used in the research and applications [1, 2, 3].

LBM (lattice Boltzmann method) is a very effective fluid dynamics calculation method, which is simple in its algorithm, can deal with complicated boundary, and has good parallelism and higher computing intensiveness. It’s very suitable for large-scale parallel computing. By using CUDA technology in LBM, Zheng Yankui et al. [4] implemented party cavity flow model with accelerated computing, its final speed is 50 times faster than that of the CPU. Zhang Yun et al. [5] implemented MRT (muti-relaxation-time) model with accelerated computing, they obtained the speed of approximately 60 times over that of the CPU. J. Tölke and M.Krafczyk[6] implemented three-dimensional D3Q13 lattice Boltzmann model with accelerated computing, they obtained an efficiency gain of up to two orders of magnitude with respect to the computational performance of the CPU.

This paper mainly introduces the method of implementing parallel computing in LBM based on LBGK model on GPU with CUDA technology and validates the feasibility and computing performance of the LBM parallel computing on GPU by using the plane Poiseuille flow as a test case.
2. CUDA Programming Model

GPU is a multi-core processor which supports parallel computing and multithreading. The hardware architecture supporting CUDA technology can be found in literature [7]. In the CUDA programming model, CPU is regarded as a host for controlling the whole serial logic and the task scheduler of a program, while GPU a coprocessor or device which performs the parallel computing parts of the program. As shown in Figure 1, a CUDA program is split into the host code that is serially executed in CPU and the device code that is parallely executed on GPU. The device code is organized into a kernel in the CUDA program, which is the function executed in concurrent threads on GPU. A thread is the basic unit of the concurrent execution, a certain number of threads are grouped into thread blocks which execute the same instructions on different data, all of the same thread blocks are grouped into a grid, and a grid is a kernel in a CUDA program. When the program is loaded, the grid is loaded into the GPU. The thread blocks are assigned to corresponding SMs (streaming multiprocessors) and the threads of the thread block are executed parallelly on all SPs (streaming processors) of the SM. Therefore, a complex computing task can be divided into some coarse-grained subproblems which can be dealt with independently, and each subproblem can be further divided into fine-grained problem until it can be processed with a single GPU thread parallely.

![CUDA programming model](image1)

A GPU thread can access many kinds of memory when a CUDA program is executed. Global memory and shared memory are the major kinds of them. The global memory is the device memory and usually large. However, it is slow in access speed and suffers high latency ranging from 400 to 600 clock cycles, and all of the threads in a grid can access it. The shared memory is the on-chip memory. Its access speed is fast, while only the threads in the same thread block can access it and the shared memory has limited storage capacity. Generally, it can be used to save shared results in a thread block. The accessing to memory is one of the important factors which influences computing performance. Especially, how to improve the performance of a global memory access seems to be very important [8].

3. The Lattice Boltzmann Method

This paper uses the Bhatnagar-Gross-Krook approximation LBM equation from Chen et al. [9]. After being discredited in the space $x$ and the time $t$, the equation is given by:

$$f_i(x+e_i,t+1) - f_i(x,t) = -\frac{1}{\tau} \left[ f_i(x,t) - f_i^{(eq)}(x,t) \right] (i = 0,1,2,\cdots,N-1),$$ (1)

where $e_i$ is a discrete velocity vector, $f_i$ the particle distribution function (PDF) with the velocity $e_i$, and $f_i^{(eq)}$ the corresponding equilibrium distribution function, $N$ the number of different speeds in the model, $\tau$ the relaxation time. The solution of Eq. (1) can be decomposed into the following steps:
Collision: \[
\tilde{f}_i(x, t) - f_i(x, t) = -\frac{1}{\tau} \left[ f_i(x, t) - f_i^{\text{eq}}(x, t) \right]
\] (2)

Propagation: \[
f_i(x + \text{e}_i \Delta t, t + \Delta t) = \tilde{f}_i(x, t)
\] (3)

where \( f_i \) is the PDF before the collision and \( \tilde{f}_i \) the PDF after the collision. In our simulations, the D2Q9 model with nine velocity directions on two-dimensional (2D) square lattice (shown in Figure 2) is used, and the corresponding particle equilibrium distribution function is given by:

\[
f_i^{\text{eq}} = \rho \omega_i \left[ 1 + \frac{3}{c^2} \left( \mathbf{e}_i \cdot \mathbf{u} \right) + \frac{9}{2c^2} \left( \mathbf{e}_i \cdot \mathbf{u} \right)^2 - \frac{3}{2c^2} \mathbf{u}^2 \right],
\] (4)

where \( \omega_i \) is the weighting factor corresponding to direction i, and \( \omega_0 = 4/9 \), \( \omega_i = 1/9 \) for \( i = 1, 2, 3, 4 \), \( \omega_i = 1/36 \) for \( i = 5, 6, 7, 8 \), \( c \) is the benchmark speed, and generally its value is 1. The macroscopic velocity \( \mathbf{u} \) and density \( \rho \) are defined as:

\[
\rho = \sum_i f_i, \quad \mathbf{u} = \left( \sum_i f_i \mathbf{e}_i \right) / \rho
\] (5)

Boundary conditions are vital importance in the application of LBM, which may significantly affect the accuracy and stability of the computations. The boundary treatment method with two-order precision proposed by Filippova and Hanel[10] is applied in this paper.

4. LBM Implementation with CUDA

4.1. Parallel Data Structure Layout of D2Q9 Model with CUDA

It is unavoidable to use global memory to store the particle distribution functions (PDFs) of all the lattice nodes because the number of the lattice nodes is huge in the simulation flow field and shared memory is limited. Global memory has high latency and no cache, so the performance of the global memory is very important for GPU. In order to access the global memory with higher efficiency, global memory accesses should be coalesced whenever it is possible. The so-called coalescent access is that: if threads within a half-warp (16 threads) of a thread block access a contiguous segment of 32, 64 or 128 bytes in the global memory, then, the 16 times accesses to the global memory will be coalesced into one time (the compute capability of GPU must be 1.2 and higher), such way will improve the efficiency of the access to global memory greatly, otherwise the accesses will be split into several times, and the efficiency of the access to global memory will be reduced remarkably [7]. So the layout of the data structure has to be designed in a way that the memory bandwidth is fully exploited. To use
coalescence, we create a one-dimensional array for each propagation direction of the particle, thus, 9 directions have 9 arrays. Each of these arrays stores the PDFs of all the lattice nodes that propagate in a certain direction. For each array, the lattice nodes are assigned consecutively along the x axis direction first, and then, along the y axis direction (see Figure 3). If each GPU thread accesses the PDF of one lattice node that propagate in a certain direction, the addresses of the global memory accessed by the threads within a half-warp of a thread block is in the contiguous segment of 128 bytes (double precision calculation is used, each data has 8 bytes, and a half-warp has 16 threads, therefore, 16×8 = 128 bytes). The half-warp threads can finish the access to the global memory in one time and achieve the coalesced access.

4.2. The Implementation of Kernel Based LBM

The process of LBM parallel computing on GPU is shown in figure 4. First, we initialize the information of the lattice nodes (including simulation flow field size, macroscopic density and velocity, and initial values of the PDFs, etc), and allocate storage space on the GPU global memory according to the layout method of the data structure in Sect.4.1. 9 one-dimensional arrays for the current time PDFs that propagate in 9 directions are allocated to the global memory. In addition to allocating another 8 one-dimensional arrays (the PDFs of the direction 0 remain still when particles propagate, it does not need to allocate an exchange array) that have same structure and size with the current arrays for exchange, the data of current arrays and the exchange arrays are exchanged at each time step to avoid overwriting problems. Then, we copy the initial values of the PDFs from the host to the current arrays in the global memory. After entering the iteration process, we make a unique association between each GPU thread and a lattice node with the threads ID. Each thread will concurrently process the collision, the boundary condition and the propagation of the lattice node. Then, the whole iterative process is repeated until the time loop is over. At the final step, the computed results are copied back to the host memory. In our implementation, GPU parallel computing mainly include three kernels: particle collision, boundary conditions treatment and particle propagation.

![Flow chart of LBM computation on the GPU](image)

Figure 4. Flow chart of LBM computation on the GPU

When we execute particle local collision according to data layout in Sect.4.1, the particles are
independent from each other, the threads within a thread block accessing the PDFs in the global memory are consecutive, which can achieve coalesced access. But in the particle propagation stage, the PDFs of each particle are transferred to its neighboring lattice nodes along 9 directions, the threads within a thread block accessing the PDFs in the global memory may be inconsecutive, which would result in uncoalesced access, and reduce the performance. So it’s very important to improve the performance of the particle propagation kernel. The literature[11] puts forward a way that combines the particle collision with the propagation into a kernel LBCollProp, which achieves the particle propagation within a thread block by using shared memory, and, in global memory, they use another kernel LBExchange to achieve the propagation of the particles on the border of a thread block which can not propagate by using shared memory. Although using shared memory for propagation can improve the performance to a certain extent, it may be restricted because the capacity of shared memory is limited, each GPU thread compute the PDFs of a row lattice nodes in the kernel LBExchange, and the computation of each lattice node in a row is serial that generally reduces the number of parallel computation threads. Furthermore, the additional cost caused by uncoalesced memory access arising from the kernel LBExchange is also considerable[12].

We separate the collision and the propagation into a collision kernel and a propagation kernel respectively, and all the access to memory is operated in the global memory. In the collision kernel, each GPU thread is mapped to a lattice node with a threads ID, and each thread computes the PDFs of the corresponding lattice node with the formula (2) and (4).To avoid overwriting, the computed results are stored in the exchange arrays directly. In the propagation kernel, each GPU thread processes one lattice node, and the PDFs stored in the exchange arrays are concurrently written back in the current arrays according to the rules of the propagation (see the formula (3)). The propagation process of the PDFs in the global memory are shown in figure 5 (where N stands for the lattice node, the gray grids stand for the lattice nodes of the bottom boundary, the dotted arrow represents the propagation rules of the PDFs which propagate in certain direction, and the solid arrow indicates the PDFs that are parallelly propagated through exchange array).

**Figure 5.** The parallel propagation of the PDF in global memory

In the horizontal direction, as shown in figure 5 (a), the PDFs of direction 1 propagate from left to right. The last lattice node of each row is the outlet node. To avoid the particle flowing out of the boundary, the periodical boundary conditions are used, In the global memory, the PDFs of position k of row n in the exchange array are written to position k+1 of row n in the current array, and then the PDF of the last position of row n in the exchange array is written to the 1st position of the same row in the current array. Through this approach, the propagation of the PDFs in direction 1 (see the solid arrow in the figure) can be completed. It is the same for the propagation of the PDFs in direction 3.

In the vertical direction of the propagation, as shown in figure 5 (b), the PDFs in direction 2...
propagate upward. In the global memory, we write the PDFs of position k of row n in the exchange array to position k of the same row in the current array. Thus, the propagation of the PDFs in direction 2 can be achieved. It is the same for the PDFs of direction 4.

In the northeast direction of the propagation, as shown in figure 5 (c), the PDFs of direction 5 will propagate to the neighboring lattice node in the northeast. In the global memory, we write the PDFs of position k of row n in the exchange array to position k+1 of row n+1 in the current array. The last lattice node of each row is an outlet node, therefore, we write the PDFs of the last position of row n in the exchange array to the 1st position of row n+1 in the current array by using the periodical boundary conditions. Thus, the PDF of the direction 5 propagation can be completed. It is the same for the PDFs of direction 7.

In the northwest direction of propagation, as shown in figure 5 (d), the PDFs of direction 6 will propagate to the neighboring lattice node in the northwest. In the global memory, we write the PDFs of position k of row n in the exchange array to position k-1 of row n+1 in the current array. The first lattice node of each row is an inlet node, thus, we write the PDFs of the first position of row n in the exchange array to the last position of row n+1 in the current array by using the periodical boundary conditions. In this way, the PDF of the propagation in direction 6 can be completed. It is the same for the PDFs of the direction 8.

In the process of propagation, all the threads within a thread block can read data from the exchange arrays in the consecutive global memory segment, thus the read access is fully coalescent, and the writing of data to the current arrays is also coalescent except the write access of the lattice node in the inlet and outlet of each row.

The kernel for boundary conditions mainly computes the unknown PDFs of the lattice nodes which are streamed from the solid boundary lattice node to the flow lattice node on the boundary. For the bottom boundary, the PDFs of the lattice node which propagate in direction 2, 5, and 6 are unknown, we use one GPU thread to process one lattice node, concurrently compute the PDFs of those lattice nodes which propagate in direction 2, 5, 6 with the method proposed by Filippova and Hanel, Then, they are written to the position of row 0 in the corresponding exchange arrays (see the gray grids in figure 5). Through the same method, the unknown PDFs of those lattice nodes which propagate in direction 4, 7, 8 on the top boundary can be computed, and then written to the position of the last row in the corresponding exchange arrays. In this kernel, all the threads only need to access the consecutive global memory segment, therefore it is a fully coalesced access.

5. Experimental Results

The experiments are carried out with a Intel Core i5-760 quad-core CPU that has a clock rate of 2.80GHz, and a NVIDIA GeForce GTS 450 graphics card equipped with 1GBytes global memory which has 24 multiprocessors of which each consists of 8 stream processors and is totaled 192 stream processors at a clock rate of 1.57GHz. The operating system is Windows 7.

In order to validate the feasibility and performance of the parallel computing of LBM on GPU, the plane Poiseuille flow is used as the experimental case. The simulation results obtained with CPU and GPU are compared, and the MLUPS (Million Lattice node Updates Per Second) and the speedup of the GPU related to the CPU are investigated by using the thread block with 8×8 threads, double precision floating point number and different mesh size at the 100000th iteration. Figure 6 shows the comparison of the simulation results run on CPU and GPU with different mesh sizes. Both results have good agreement. The results indicate that parallel computing of the LBM on GPU is completely feasible. Table 1 lists the MLUPS and the speedup run on CPU and GPU respectively with different mesh sizes. When the mesh size is 32×64, the speedup is only 44 times. If each thread processes a lattice node, each thread block has 64 threads, and then there are 32 thread blocks totally. The GTS 450 GPU has 24 multiprocessors and each multiprocessor can be allocated less than 2 thread blocks. Therefore, only when each multiprocessor is allocated more than 2 thread blocks, then the GPU can give full play to the parallel computing performance according to literature [7]. Along with the increase of the mesh size, the speedup also increases. When the mesh size is 64×128, 128×256 and 256×512 respectively, their corresponding speedups are 76, 93 and 102 times. It also indicates that the number of the thread blocks increases with the increase of the mesh size, each multiprocessor can be allocated with more active thread blocks, and the number of the threads, which computes simultaneously, increases largely. It can hide the high latency brought by the global memory and improve the
performance. However, the speedups are lowered to 100 and 98 times when the mesh size is 512×1024 and 1024×2048 respectively. It also indicates that the speedup can be reduced because of the limitation of GPU hardware resources when the mesh size increases to certain extent. The accelerating performance is still very significant comparing to CPU.

![Simulation results on CPU and GPU on the 100000th time steps](image1)

**Figure 6.** Simulation results on CPU and GPU on the 100000th time steps

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>CPU time(s)</th>
<th>GPU time(s)</th>
<th>MLUPS(CPU)</th>
<th>MLUPS(GPU)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32×64</td>
<td>308.339</td>
<td>7.002</td>
<td>0.664</td>
<td>29.249</td>
<td>44</td>
</tr>
<tr>
<td>64×128</td>
<td>1277.790</td>
<td>16.860</td>
<td>0.641</td>
<td>48.588</td>
<td>76</td>
</tr>
<tr>
<td>128×256</td>
<td>5170.940</td>
<td>55.435</td>
<td>0.634</td>
<td>59.111</td>
<td>93</td>
</tr>
<tr>
<td>256×512</td>
<td>21120.500</td>
<td>207.728</td>
<td>0.621</td>
<td>63.098</td>
<td>102</td>
</tr>
<tr>
<td>512×1024</td>
<td>83960.300</td>
<td>838.451</td>
<td>0.624</td>
<td>62.531</td>
<td>100</td>
</tr>
<tr>
<td>1024×2048</td>
<td>336771.410</td>
<td>3427.350</td>
<td>0.623</td>
<td>61.189</td>
<td>98</td>
</tr>
</tbody>
</table>

### 6. Summary

In this Paper, parallel computation of the LBM based on LBGK model on GPU with CUDA technology is implemented. In the simulation of the plane Poiseuille flow, the results obtained with both CPU and GPU have good agreement with each other. The results indicate that the parallel computation of the LBM on GPU is completely feasible. The computing performance run on CPU and GPU is compared in the simulations, in which the highest speedup obtained with GPU can be 102 times faster than that obtained with CPU. The results also show that using the GPU can greatly accelerate the LBM computation, which provides a very effective way to solve the complex problem of the modern computational fluid dynamics on a low cost computer.

### 7. Acknowledgements

This work has been supported by the Natural Science Foundation of China under Grants No. 11162002 and the “Bagui Scholar” Project Special Funds.
8. References


