Journal of Petroleum Science and Engineering 156 (2017) 546–552

Contents lists available at ScienceDirect

Journal of Petroleum Science and Engineering

journal homepage: www.elsevier.com/locate/petrol

GPU-accelerated volumetric lattice Boltzmann method for porous media flow

Senyou An [a,b], Huidan(Whitney) Yu [b,*], Jun Yao [a,**]

[a] School of Petroleum Engineering, China University of Petroleum, Qingdao, 266580, China
[b] Department of Mechanical and Energy Engineering, Indiana University-Purdue University, Indianapolis, IN, 46202, USA

ARTICLE INFO

Keywords:
Porous media
GPU parallel
Volumetric lattice Boltzmann method
Digital core
Petroleum

ABSTRACT

The volumetric lattice Boltzmann method (VLBM) has been recently developed and validated for dealing with flows in complex geometries. To reveal the intricate and arbitrary porous media skeleton, VLBM categorizes the computational domain into fluid, solid, and boundary cells by introducing a volumetric parameter $P(\vec{x})$, through which the lattice Boltzmann equations are self-regularized. As a result, the no-slip bounce-back boundary condition at the inter walls is integrated in the streaming term. Since its data structure is aligned and kernel pattern is clear, VLBM is ideally suited for GPU parallelization. Using the $P(\vec{x})$ in the streaming operation, branch diverse can be effectively decreased. In this paper, we use several optimization methods, such as memory arrangement and kernel design, to maximize the performance of parallelization for VLBM. As an application, we simulated petroleum flow in a digital sandstone with two resolutions, 256^3 and $256^2 \times 512$, and evaluated its permeability. The best parallel performance reaches 808.7 MLUPS (Million Lattice Updates Per Second), which is 1421.3-times speedup compared with the serial computation with allocated memory.

1. Introduction

The lattice Boltzmann method (LBM) (Benzi et al., 1992; Chen et al., 1992) has become a popular alternative to traditional Navier-Stoke (NS) equation solvers (e.g. finite element method, finite volume method), especially for incompressible and time-dependent flow (Chen and Doolen, 1998; Aidun and Clausen, 2010). As a heritage from cellular automaton, the LBM simulates fluid dynamics via prescribed discrete kinetic equations for time evolution of discrete particle density distribution functions due to molecular interaction. The macroscopic properties such as velocity, pressure and wall shear stress, are the macro-scale reflection of particle distribution. Mathematically, the incompressible lattice Boltzmann equation can recover NS equations based on BGK(Bhatnagar-Gross-Krook) collision approximation (Bhatnagar et al., 1954) and Chapman-Enskog technique (Chapman and Cowling, 1970) to the second-order accuracy in both space and time (Chen and Doolen, 1998).

In the traditional LBM, fluid particles are set on the node points and the computational domain is usually characterilized by 0 and 1, representing fluid and solid. The particle distribution functions represent the corresponding cell's density layout linked with momentum distribution. In the iterative evolution, particles' collisions occur in host cells, and then the particles stream to adjust grids along their velocity directions. When a lattice cell is cut by arbitrarily curved boundaries, either a fluid or solid node has to be determined via the volume fraction of solid. Such a treatment may alter the real flow domain, which can be significantly in accurate in porous media flow. To improve the accuracy, the computational resolution must be very fine, causing demandingly high computation cost. Unstructured mesh may ease the computation demand (Peng et al., 1999; Li et al., 2005), but it is not applicable for porous media flow because there will be difficulties in small isolated areas or extreme tip points in random porous structure (Aavatsmark et al., 1998). The unstructured mesh may also weaken those strong points of structured LBM in GPU parallelization (Qian et al., 1995; Feichtinger et al., 2011). Recently, Yu, et al. developed a mass-conserved volumetric lattice Boltzmann method (VLBM) (Yu et al., 2014), in which fluid particles are uniformly distributed in each lattice cell. The computational domain are categorized through fluid, solid, and boundary cells by introducing a volumetric parameter $P(\vec{x})$, defining the solid fraction in each cell. The volumatic lattice Boltzmann equations are self-regulated through $P(\vec{x})$. In VLBM, the no-slip bounce-back boundary condition at the inter walls is integrated in the streaming term. The introduction of volumatic

* Corresponding author.
** Corresponding author.
E-mail addresses: whyu@iupui.edu (H. Yu), rcogfr_upc@126.com (J. Yao).

http://dx.doi.org/10.1016/j.petrol.2017.06.031
Received 3 May 2017; Received in revised form 13 June 2017; Accepted 14 June 2017
Available online 17 June 2017
0920-4105/Published by Elsevier B.V.

representation makes an enormous contribution to accurately describe complex boundaries with no compromise of fine resolution, which is great for porous media flow.

The VLBM is ideally suitable for GPU parallelization. Previously, Wang, et al. parallelized VLBM for simulating blood flow in human arteries (Wang et al., 2015). The performance was about 100 MLUPS (Million Lattice node Updates Per Second). This parallel scheme ended up with 10 MLUPS for crude oil flow in digital stone, implying a need of more advanced parallel schemes. In this work, we parallelize the VLBM by highly memory-efficient technologies, including memory arrangement and kernel structure design. First, we have adopted the modified tiling algorithm from Tran N P, et al. (Tran et al., 2015) to minimize the effect of the uncoalesced accesses. Second, we develop a new allocation method for CPU memory to guarantee enough continued space for reading geometry data. Third, we optimize the parallelization by removing branch divergences, arranging the register usage and combining streaming and collision into a kernel to maximize the acceleration of computation. To demonstrate the application, we simulate porous media flow in a digital core by using the GPU-accelerated VLBM.

The remainder of the paper is organized as follows. Section 2 introduces the mathematical formulation of VLBM. The high-efficient GPU parallelization for VLBM is presented in Sec.3. An application about crude oil flow in porous media is studied in Sec.4. Finally, Sec.5 provides a summary discussion and concludes the paper.

2. Volumetric lattice Boltzmnn method

The VLBM has been introduced in the previous paper in detail (Yu et al., 2014). Herein, we just cite the foundational concepts and equations for better comprehension of parallel computation. VLBM is designed to accurately and conveniently deal with boundaries by defining solid occupation function $P(\vec{x})$. Just like Fig. 1, extracted for vertical direction of the D3Q19 model, $P(\vec{x}) = 0$ is used to represent fluid domain, and the shaded (solid) zone means $P(\vec{x}) = 1$. When the cell is crossed by the boundary line, it will have partial fluid and partial solid. Correspondingly, the solid occupation function will be between 0 and 1. So the arbitrary structure can be described with less grids compared with traditional LBM.

The fluid particles are sited in lattice cells, as opposed to distributed on the nodes in the conventional LBM. To reflect the effect of Pvalue $P(\vec{x})$, the distribution function is defined as $n_i(\vec{x}, t) = f_i(\vec{x}, t)\cdot P(\vec{x})$ with velocity $\vec{e}_i$ occupying a lattice cell $\vec{x}_i$ at time t and it deals with the time evolution of the particle distribution function analogy to LBE.

$$n_i(\vec{x} + \vec{e}_i\Delta t, t + \Delta t) = n_i(\vec{x}, t) + \Omega_i(\vec{x}, t) \tag{1}$$

where $\Omega_i(\vec{x}, t)$ is a collision term due to the molecule particles' motion and the momentum exchange between them. The most common calculation for this part is the BGK model with a single-scale relaxation time τ, as Eq. (2). And $i = 0, 1, 2, \cdots, b$ represents predefined directions of molecular motion.

$$\Omega_i(\vec{x}, t) = -\frac{1}{\tau}[n_i(\vec{x}, t) - n_i^{eq}(\vec{x}, t)] \tag{2}$$

where the $n_i^{eq}(\vec{x}, t)$ is the equilibrium function, which is formed as:

$$n_i^{eq}(\vec{x}, t) = N(\vec{x}, t)\omega_i\left[1 + \frac{\vec{e}_i\cdot\vec{u}}{c_s^2} + \frac{(\vec{e}_i\cdot\vec{u})^2}{2c_s^4} + \frac{(\vec{u})^2}{2c_s^2}\right] \tag{3}$$

where $N(\vec{x}, t) = \sum n_i(\vec{x}, t)$ is density at the current cell, ω_i is weight fraction of ith velocity direction, and c_s is speed velocity.

To avoid confusion, streaming and collision should be separated in the calculation. After collision, temporary array "postcollision" is defined as $n'_i(\vec{x}, t)$, as the following equation:

$$n'_i(\vec{x}, t) = n_i(\vec{x}, t) + \Omega_i(\vec{x}, t) \tag{4}$$

Streaming means that the fluid particles move from the current cell to neighboring cells. The key point for VLBM is the streaming part, as it considers the fluid volume fraction of boundary cells. Only a specific fluid can stream to these cells, meaning the other part will be bounced back to the host cell. As illustrated in Fig. 1, the 11, 5 and 12 direction velocities contain two parts, particles streaming from previous cells (black solid arrow) $[1- P(\vec{x}, t)]n'_i(\vec{x}- \vec{e}_i\Delta t, t)$ and bounce-back fluid from downwind boundary cells, $P(\vec{x}+ \vec{e}_{i\star}\Delta t, t)n'_{i\star}(\vec{x}_i, t)$, as shown in following equation:

$$n''_i(\vec{x}, t + \Delta t) = [1 - P(\vec{x}, t)]n'_i(\vec{x} - \vec{e}_i\Delta t, t) + P(\vec{x} + \vec{e}_{i\star}\Delta t, t)n'_{i\star}(\vec{x}_i, t) \tag{5}$$

where $i^\star$ corresponds to the opposite velocity direction with i direction, meaning $\vec{e}_{i^*} = -\vec{e}_i$. This modified streaming process ensures that particles are advected or reflected to their appropriate places in the fluid domain, but does not introduce any extra mass.

The resulting density, velocity and pressure are obtained as follows:

$$\rho(\vec{x}, t) = \sum n_i(\vec{x}, t)/[1 - P(\vec{x}, t)] \tag{6}$$

$$\vec{u}(\vec{x}, t) = \sum \vec{e}_i n_i(\vec{x}, t) \Big/ \sum n_i(\vec{x}, t) \tag{7}$$

$$p(\vec{x}, t) - p_0 = c_s^2[\rho(\vec{x}, t) - \rho_0] \tag{8}$$

where p_0 and ρ_0 are original reference pressure and density, respectively.

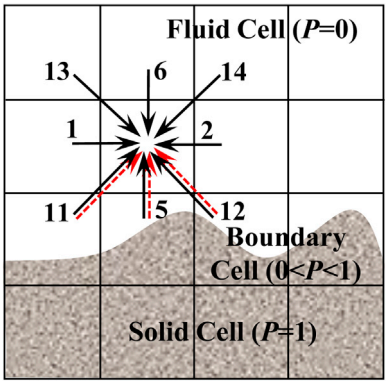

Fig. 1. Illustration of volumetric representation of cells using solid ratio P: $P = 0$ (fluid), $P = 1$ (solid), $0 < P < 1$ (boundary).

3. GPU parallelization for VLBM

One of the most important strengths of LBM is ideally suitable for GPU parallel. In this section, we introduce how to realize memory efficient parallelization for D3Q19-based VLBM and optimize it to high speed, including dynamic allocation, coalesced global memory, register arrangement, and kernel structure design.

3.1. Dynamic allocation for CPU

The limits of CPU static zone memory and GPU device memory are essential factors to block acceleration and improvement of model size, especially when the big data file exists in the data transformation between host and device. For porous media flow, structure information and hydrodynamic parameters (Pvalue, density, velocity and pressure), are essential arrays to be defined for output in CPU host. Distribution function arrays are applied, initialized and updated in GPU directly. Assuming the model size is $512 \times 512 \times 512$ and all the data is float type,

the needed memory for these four arrays will be exactly 2GB, meaning the host should have more than 2GB to store this data as other few existed variables. But if we need calculate wall share stress, e.g. in biomedical field, or if the distribution function is also defined in CPU for previous processing, the needed memory will be larger than 2GB. There are typically file and static memory limits imposed by either an operating system or system administrator that cap at about 2 GB in the host. Nowadays, a single GPU card is capable of storing anywhere from 3 to 12 GB of data in memory, and multi-GPU technology will multiply increase this memory. In order to fully utilize this space, the CPU part of this algorithm relies on dynamic (heap) memory as opposed to statically allocated (stack) memory.

If an uncompiled code were to attempt to use static allocation to utilize the full capacity of a GPU, it would run into the 2 GB size limit as the executable created by the compiler would either need to be larger than the file size limit or exceed the static memory limit. Dynamic allocation allows the operating system to provide as much memory as what is available to provide by the CPU RAM. So the algorithm can fully utilize the memory capacity of the GPU device when the dynamic memory is declared. For the purposes of data copy between CPU host and GPU device, the transmission arrays should have logically contiguous memory. The [*new*] keyword is a provided tool in C++ to allocate logically contiguous dynamic memory. The [*new*] keyword is limited, however, to dynamically allocating one dimensional arrays. We create a class in C++, which would dynamically allocate a contiguous block of memory for multidimensional arrays using this keyword. The class would then internally translate the request from the 24 dimensional element to the equivalent request in 1 dimension. This class is supposed to easily pass its underlying data to the GPU and then easily retrieve the information from the GPU and to manage allocation and deallocation of memory in an easy way, the outline is shown as Fig. 2.

The Array2, Array3, and Array4 template classes were created to meet these requirements. The template arguments is provided to specify the number of values in each dimension like: *Array3<float, NX, NY, NZ>Pvalue*, where each template argument after the first, such as NX, can be a mutable unsigned long long integer. The first argument can be changed to a double or whatever underlying data type is desired by the user. Internally, the Array N family of classes manages the dynamic memory by properly overloading the copy-constructor, destructor, and assignment operators for the class. In order to transfer the class's underlying data to and from the GPU, the public method *data*() is provided. This method returns a pointer to the start of the contiguous 1D array that can be read into or from a GPU card using the appropriate memcpy command. Finally, array-like access is provided by overloading operator [ ] at various levels so that the syntax and behavior is identical to what one would expect from statically allocated multidimensional arrays. The *ArrayN* family of classes should provide programmers who have been using regular, statically allocated multidimensional arrays a quick and easy way to update their programs to utilize the full memory capacity of the GPU cards.

```
//! copy constructor
Array( const Array & other )
{
  allocate();
  assign(other.m_data);
}

//! assignment operator
Array & operator=( Array other )
{
  swap(other);
  return *this;
}
```

Fig. 2. The declaration of arrays in CPU host.

3.2. Coalesced global memory

Herein, we use the 1-D array to implement data organization for multi-dimensional arrays to avoid extra de-referencing. For geometry structure, the array needs $N_x \times N_y \times N_z$ elements (N_x, N_y and N_z are width, height and depth of the grid). The distribution function $n_i(\vec{x}, t)$ and post-collision array $n'_i(\vec{x}, t)$ both have $N_x \times N_y \times N_z \times 19$ elements.

This data is declared in the global memory, which belongs to the off-chip memory (pink block), as shown in Fig. 3. The global memory is a storage area available for all the thread blocks, through which data can realize two-direction communication, and CPU can also input and output information for the device. Just like the heap memory in host, global memory allocations can persist throughout the duration of the application until the execution of free order or the end of the program. The bandwidth of global memory, however, significantly limits calculation speed. In present technology, arranging data to apply bandwidth as effectively as possible is an important step in optimization. For these arrays, there are usually two common data organization schemes:

(1) Array of Structure (AoS): 19 distributions of each cell occupy 19 consecutive elements of the array. Its plated form is $n_i[z\star N_x\star N_y\star 19+ y\star N_x\star 19+ x\star 19+ i]$, and x, y and z are corresponding positions on the grid, for example the z direction position can be calculated with this equation: $z = blockIdx.z\star blockDim.z+ threadIdx.z$. This scheme is preferable for CPU series parallelization, as the nodes will be implemented one by one under the control of the main thread.

(2) Structure of Array (SoA): the value of one distribution of all cells is arranged consecutively in the memory. Herein, we use 19 arrays to store the different data in the 19 directions of the D3Q19 model, just like the direction 1, all n_1 s in the domain being stored in an array. The structures of AoS and SoA are shown in Fig. 4.

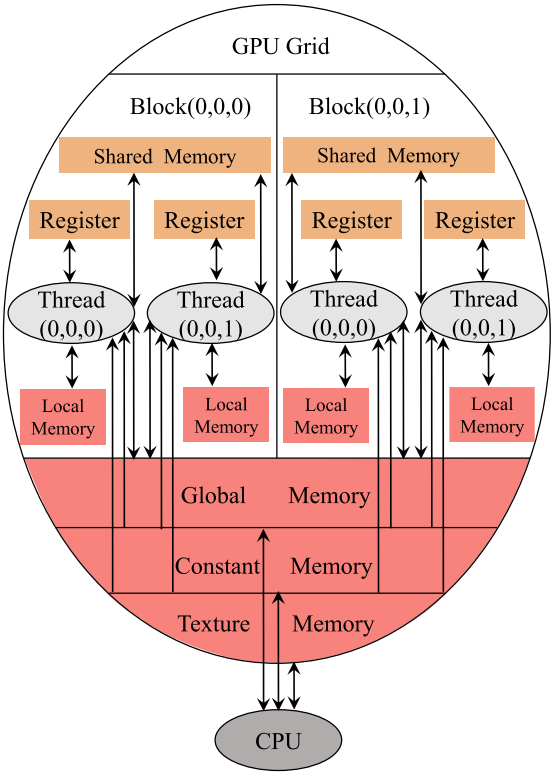

Fig. 3. The illustration of GPU device memory.

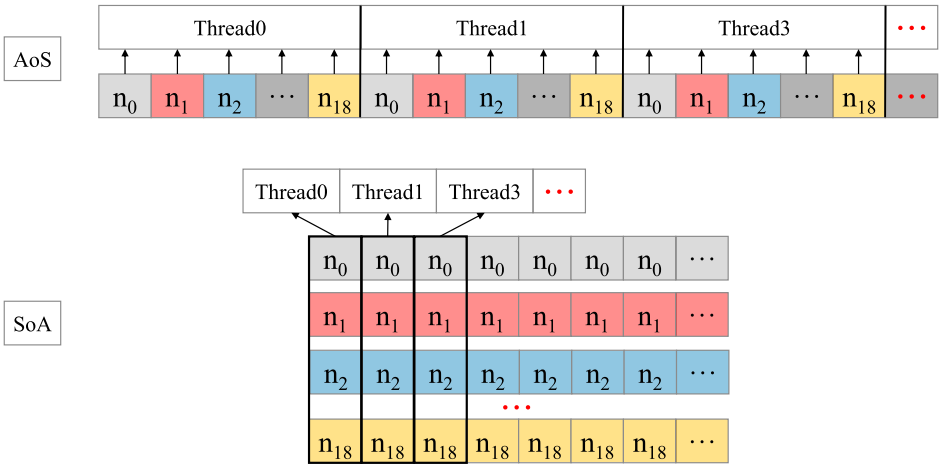

Fig. 4. The AoS and SoA schemes.

The most effective way to access the global memory is to ensure that all threads in a block can access a consecutive memory location (Luitjens, 2011). In the GPU, a warp is a unit made from 32 threads to access global memory. If the memory address is continued and the visiting memory of a warp is lower than the specific facility limit, the fast access can be achieved once. So, the SoA scheme is obviously suitable for the GPU grid because it guarantees that different threads call for the consequent addresses (Delbosc et al., 2014).

3.3. Kernel design for GPU structure

As introduced in the previous section, the global memory belongs to the off-chip memory, which has far lower accessing speed than the on-chip memory (Panda et al., 2000). In most cases, accessing a register consumes zero clock cycles per instruction (Reese and Zaranek, 2012), compared to 400–800 cycles for global memory (Power et al., 2014). So efficient utilization of cache space is extremely important in modern embedded system applications based on processor cores. As shown in Fig. 5, we try to decrease the visiting time to the global memory by combining the streaming operation, hydrodynamic update and collision operation into a kernel opposite two separated kernels.

In the evolution kernel, the needed parameters are defined in the registers, and assigned values from the global memory. All subsequent calculations are based on the variables defined in the registers. The global memory arrays are updated before the end of the current kernel as the register variables will be freed at the same time as the kernel ends. In this part, only variables can be stored in the registers, which means that the array should be separated, as structures or arrays are typically

```
__global__ void lbm_evolution( )
{
int tx = blockIdx.x*blockDim.x + threadIdx.x;
int ty = blockIdx.y*blockDim.y + threadIdx.y;
int tz = blockIdx.z*blockDim.z + threadIdx.z;
int tid = tx*NY*NZ + ty*NZ + tz;
float Pf = D_Pvalue[tid];
int id1, id2, id3, ⋯ , id18;
float f0, f1, f2, f3 , ⋯ ,f18;
float ux_r, uy_r, uz_r, den_r;
//  Load distribution functions from global memory to registers and stream.
//  Calculate the needed neighbor ID.
id1 = (tx + 1)*NY*NZ + ty*NZ + tz;
 ⋯
id18 = tx*NY*NZ + (ty - 1)*NZ + tz - 1;
//  Streaming based VLBM streaming equation.
f0 = f0.old[tid];
f1 = (1 - Pf)*f1.old[id2] + D_Pvalue[id2] * f2.old[tid];
 ⋯
f18 = (1 - Pf)*f18.old[id15] + D_Pvalue[id15] * f15.old[tid];
//  Compute velocity and density.
//  Collision
f0 = (1 - 1 / tau)*f0 + equilibrium(0, den_r, ux_r, uy_r, uz_r) / tau;
 ⋯
f18 = (1 - 1 / tau)*f18 + equilibrium(18, den_r, ux_r, uy_r, uz_r) / tau;
//  Write all the distribution functions to global memory
//  Write velocity and density to global memory
}
```

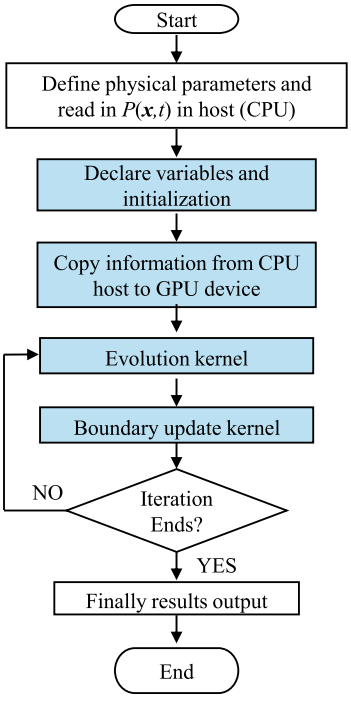

Fig. 5. Flow chart of GPU parallelism of VLBM (right) and the corresponding code for evolution kernel (left).

addressed in local memory. The local memory is an abstraction of global memory, not a physical memory type. Its scope is local to the thread and it resides off-chip, which makes it as expensive to access as global memory. Furthermore, the register memory is small, usually 32 bits per register (the total number of registers available per block is 65,536 for Tesla K20). The compiler will also make use of local memory when it determines that there is not enough register space to hold the variables, and this process is called register spilling.

In VLBM, the streaming evolution is more complex than the traditional node-based LBM, as it requires more variables, including neighboring information in addition to local $P(\vec{x})$ (geometry information). Thus, the balance between the latency from off-chip memory accesses and the occupancy of computational resources needs to be considered. For a GPU facility with computational capability of 3.5 (e.g Tesla K20), the maximum number of threads per streaming multiprocessor (SM) is 2048; the max warps per SM is 64 and the max blocks per SM is 16. So, the maximum number of registers per thread can be calculated: $\frac{65536}{2048} = 32$. To reach the best performance, all these limiting conditions should be taken into account. The following are three common principles in the design of grid and block size.

- The number of threads in a block should be a multiple of 32, if the data structure allows.
- The size of a block should be at least 128 (2048 ÷ 16). Here, we arrange 256 threads in a block.
- The size of a grid (the number of blocks) is far lager than the number of SMs.

Under these limits, the local variables in the evolution kernel are further optimized to satisfy the register resource limit. For simple operations that can be calculated easily, we don't declare them to be the memory variables. Besides, the indexing address of distributions is calculated directly without declaration, as shown in the schematic code of Fig. 5. We can reduce the register demand to 29 (less than 32), which means that the active warp has 100% occupancy. Actually, many researches find 66% is enough to saturate the bandwidth at present level (Luitjens and Rennich, 2011). Therefore, even though bandwidth may improve in the near future, we still can use this implementation to meet it.

3.4. Automatic optimization in the VLBM

Streaming and collision are two basic terms both in the VLBM and in the traditional node-based LBM. Collision has no special design for calculation, as it has a simple self-based data structure as shown in Eq. (2). In this paper, the streaming term is designed as a *pull* model in the VLBM, and occupancy function $P(\vec{x})$ can avoid branch divergence automatically.

As introduced in Section 3.2, coalescing global memory access is of great importance to improve performance. To guarantee the memory access can achieve the best bandwidth performance, the addresses should be in a continuous 128-byte range. Usually, the LBM algorithms do the calculations in the current host cell, and the particles stream to its neighbors in the subsequent operation, as illustrated in Fig. 6. In this scheme, the read offset is aligned and the write offset is misaligned. Just like direction 1 and 2 in the D3Q19 model, the update positions are shifted to the places that do not belong to the 128-byte portion while the thread indexes do not change correspondingly. The other method is the pull scheme, which reverses the push structure by exchanging the order of reading and writing for cells. First, the particle distribution functions in adjacent cells are read and streamed to the current cell. Then the collision processing is implemented, and the updated data is written to the various directions in the host cell. For pull structure, alignment occurs in writing, and the reading is unaligned. Since considering the cost of the uncoalesced reading is smaller than the cost of the uncoalesced writing, the pull scheme is adopt for the VLBM, which is reflected in Eq. (5).

Warp is the actual unit being controlled in the GPU SM. The coed is compiled together for the 32 kernels in a warp and executed with different input data. If the treads diverge in different directions by using [*if*] or other judging keywords, the performance will be significantly affected. Under this condition, the different members will choose different paths, and these paths are serialized to wait for each other. So the branch divergence should be avoided to realize the real parallelism.

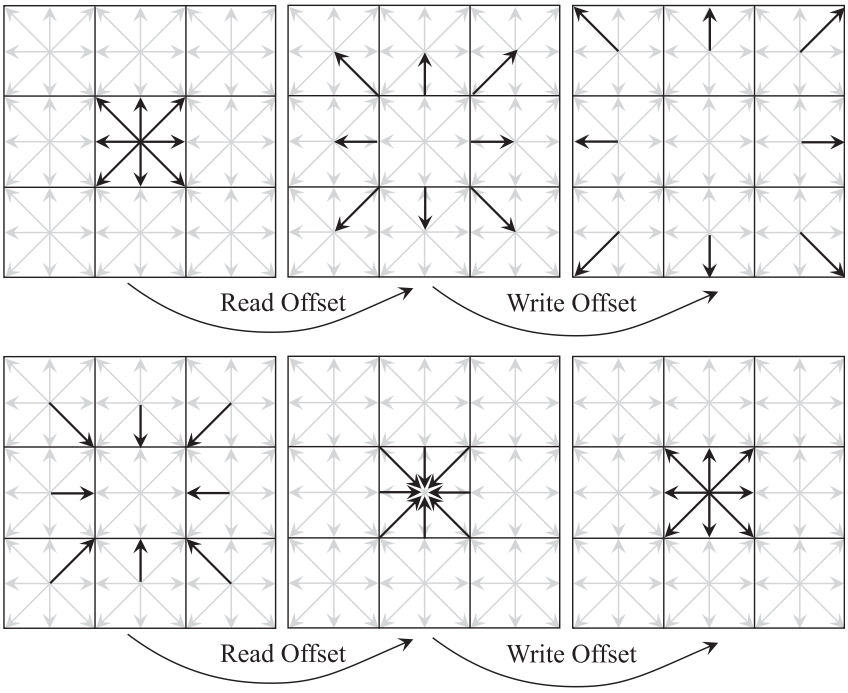

Fig. 6. In the shift algorithm, lattice extension and offsets for memory operations are utilized. There are two offsets: one for reading and one for writing. The upper one is push scheme, and lower one is pull scheme.

Fig. 7. Three dimensional gray image (a) of core with resolution 3.7 m/pixel.

For porous media, the first operation is to separate solid and fluid cells within the node-based LBM. Otherwise, the calculation will occur in the solid cells, which will result in complete failure. To solve this problem, some researchers proposed extracting the fluid cells and recording the positions of their neighbors. However, the fluid proportion cannot be too big for this method (usually less than 35%) to ensure that the cycles' consumption of position array access is less than the fluid cell judgment. We do not need these special operations for the VLBM because of the existence of function $P(\vec{x})$. Nothing can steam into the solid cells as the upwind distribution function should multiply the current cell's fluid section. In addition, the distribution functions in solid cells are all zero, which won't bring extra particles to fluid cells.

If we use specific conditions to find boundary layers, it also will induce branch divergence. In order to avoid using if-statements, one solid layer is attached per face to set the seal boundary perpendicular to the flow direction. The inlet and outlet boundaries are updated based on non-equilibrium processing. This boundary-updated step is realized in another kernel.

4. Application study

A sandstone is used in this paper to demonstrate the efficiency of GPU parallelism for the VLBM. One cylindrical core plug is drilled from the stone with a length of 4 cm and a diameter of 10 mm. Due to different phases having various X-ray absorption rates, the digital core is described by gray value between 0 and 255 after the CT scanning. Then the sub-volume data ($256 \times 256 \times 256$ and $256 \times 256 \times 512$) is cut from the original image, as shown in Fig. 7. The step between the raw image and flow simulation is 3D structure reconstruction, which is a complex operation for conventional NS equation solvers (e.g. COMSOL, FLUENT). When the VLBM is used to solve these equations, regular mesh is obtained without other extra smoothing or unture representational operations. For seamless connection with the VLBM, the distance field is calculated based on level-set equation (Wang et al., 2011; Thömmes et al., 2009; Balla-Arabe et al., 2013), which is modified by introducing multi-seed initial contours to guarantee that the segmentation can be as close as possible to the real image. After the segmentation, the $P(\vec{x})$ field is calculated using the distance field and local refining method (Yu et al., 2014).

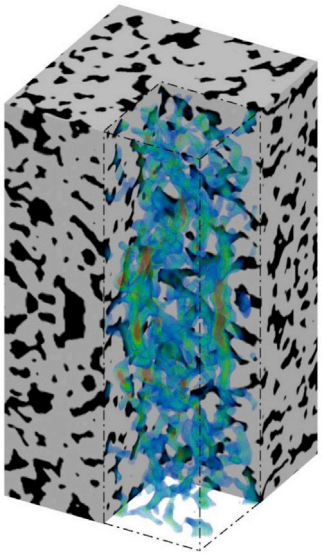

Fig. 8. The velocity field distribution in digital core model.

Table 1
The results of GPU acceleration.

Model Size	GPU	MLUPS(GPU)	MLUPS(CPU)	Acceleration
$256 \times 256 \times 256$	K20	450.7	0.680	662.8
$256 \times 256 \times 256$	K40	749.1		1101.6
$256 \times 256 \times 512$	K40	808.7	0.569	1421.3

The constant pressures are set at inlet and outlet boundaries to simulate the reservoir development method with constant pressure difference. Specifically, the pressure difference is $\Delta P = 10^4 Pa$ in this algorithm. The used crude oil has following properties: the API gravity is 25.7, the dynamic viscosity is $13.5 mPa{\cdot}s$ and the fluid is incompressible. Then, the fluid flow is simulated in this domain to obtain the velocity field as shown in Fig. 8. Correspondingly, the permeability K can be calculated based on Darcy's equation (An et al., 2016) $K = Q\mu L/A\Delta P = 113.5 mD$.

We carried out the computation on two different facilities: one has a NVIDIA Tesla K20 GPU card, which has 2496 CUDA cores with 706 MHz clock frequency and 5 GB globe memory; the other owns a NVIDIA Tesla K40 GPU card with 2880 CUDA cores, 745 MHz clock frequency and 12 GB globe memory from Maverick in XSEDE. To evaluate the GPU parallel performance, the serial computation is performed with a Dell C8220 CPU, which has 16 computing cores with 2.7 GHZ and 32 GB DDR3 Random Access Memory (RAM).

The calculation performance is examined via MLUPS (Million Lattice node Updates Per Second). As the static memory is not enough for the pure C algorithm, the functional keyword [*new*] is used to apply for dynamic memory in heap memory, which lowers the access speed. For the $256 \times 256 \times 512$ model, the average speed is 61 steps per hour. Correspondingly, the MLUPS is 0.569. When the code is paralleled by the GPU introduced in previous content, the MLUPS is 808.7 with a speed of 24.1 steps/min, accelerating the CPU serial computation by 1421.3 times, as shown in Table 1. Wang et al. proposes GPU parallelization for the VLBM and applies it to carotid hemodynamics based on a Geforce GTX 780 GPU, in which the MLUPS is 129.5 for $133 \times 125 \times 352$ model (Wang et al., 2015). Their inlet and outlet boundaries are based on velocity profile, which is more simple than the no-equilibrium pressure boundary used in our algorithm. In addition, the Geforce GTX 780 GPU is better than the K20 in calculation performance. Therefore, there is an obvious improvement in the GPU acceleration in our optimized code.

5. Summary and discussion

We have presented a high-efficient GPU parallelization for VLBM and applied it to a simulation of crude oil flow in a digital reservoir stone. For the sample with resolution $256^2 \times 512$, we have achieved 808.7 MLUPS and 1421.3 times speedup to serial computation utilizing one single GPU card (NVIDIA Tesla K40). The algorithm includes the following technical features.

- The host memory allocates the continued memory in heap zone to read structure information ($P(\vec{x})$ data file).

- A tiling optimization with data layout changes is designed for coalesced global memory to overcome bandwidth limit of off-chip memory and achieve special unit access.
- A pull scheme is used for the streaming term in VLBM to allow uncoalesced Read Offset and vcoalesced Write Offset.
- The kernel organization is optimized by combining streaming evolution, collision updating and boundary refreshing into a kernel. Therefore, we can read pertinent variables once from global memory, meaning the cache can be used as effectively as possible.
- Branch divergence is avoided inherently due to the introduction of $P(\vec{x})$ in VLBM.

All these key points have been implemented in the application study, resulting in a significant acceleration. We will work on more applicable studies in the field of petroleum engineering by integrating more physical models such as interfacial dynamics for multiphase flows, fluid-structure interaction for deformable pore structure, non-Newtonian effects, etc. However, more sophisticated modeling means higher computational expense. We are currently working on the multi-GPU-card implementation for VLBM. We will focus on the critical steps for implementing asynchronous data communication between GPUs, which includes transfer between the compute nodes. Once faster computation, e.g. 2000 MLUPS, is available, the important phenomena of crude oil flow in digital stone such as relative phase permeability curve and acid reaction between fluid and solid can be accurately explored.

Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is support by (1) International Research Development Fund (IRDF) of IUPUI, (2) the National Natural Science Foundation of China (No.51490654), and (3) Program of Innovation Projects (YCXJ2016021).

References

Aavatsmark, I., Barkve, T., Bøe, O., Mannseth, T., 1998. Discretization on unstructured grids for inhomogeneous, anisotropic media. part i: derivation of the methods. SIAM J. Sci. Comput. 19, 1700–1716.

Aidun, C.K., Clausen, J.R., 2010. Lattice-boltzmann method for complex flows. Annu. Rev. fluid Mech. 42, 439–472.

An, S., Yao, J., Yang, Y., Zhang, L., Zhao, J., Gao, Y., 2016. Influence of pore structure parameters on flow characteristics based on a digital rock and the pore network model. J. Nat. Gas Sci. Eng. 31, 156–163.

Balla-Arabe, S., Gao, X., Wang, B., 2013. A fast and robust level set method for image segmentation using fuzzy clustering and lattice boltzmann method. IEEE Trans. Cybern. 43, 910–920.

Benzi, R., Succi, S., Vergassola, M., 1992. The lattice boltzmann equation: theory and applications. Phys. Rep. 222, 145–197.

Bhatnagar, P.L., Gross, E.P., Krook, M., 1954. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. Phys. Rev. 94, 511.

Chapman, S., Cowling, T.G., 1970. The Mathematical Theory of Non-uniform Gases: an Account of the Kinetic Theory of Viscosity, Thermal Conduction and Diffusion in Gases. Cambridge university press.

Chen, S., Doolen, G.D., 1998. Lattice boltzmann method for fluid flows. Annu. Rev. Fluid Mech. 30, 329–364.

Chen, H., Chen, S., Matthaeus, W.H., 1992. Recovery of the navier-stokes equations using a lattice-gas boltzmann method. Phys. Rev. A 45, R5339.

Delbosc, N., Summers, J.L., Khan, A., Kapur, N., Noakes, C.J., 2014. Optimized implementation of the lattice boltzmann method on a graphics processing unit towards real-time fluid simulation. Comput. Math. Appl. 67, 462–475.

Feichtinger, C., Habich, J., Köstler, H., Hager, G., Rüde, U., Wellein, G., 2011. A flexible patch-based lattice boltzmann parallelization approach for heterogeneous gpu–cpu clusters. Parallel Comput. 37, 536–549.

Li, Y., LeBoeuf, E.J., Basu, P., 2005. Least-squares finite-element scheme for the lattice boltzmann method on an unstructured mesh. Phys. Rev. E 72, 046711.

Luitjens, J., 2011. Global Memory Usage and Strategy, Technical Report. Technical report. NVIDIA Corporation.

Luitjens, J., Rennich, S., 2011. Cuda warps and occupancy. GPU Comput. Webinar 11.

Panda, P.R., Dutt, N.D., Nicolau, A., 2000. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. ACM Trans. Des. Autom. Electron. Syst. (TODAES) 5, 682–704.

Peng, G., Xi, H., Duncan, C., Chou, S.-H., 1999. Finite volume scheme for the lattice boltzmann method on unstructured meshes. Phys. Rev. E 59, 4675.

Power, J., Hill, M.D., Wood, D.A., 2014. Supporting x86-64 address translation for 100s of gpu lanes. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp. 568–578.

Qian, Y.-H., Succi, S., Orszag, S., 1995. Recent advances in lattice boltzmann computing. Annu. Rev. Comput. Phys. 3, 195–242.

Reese, J., Zaranek, S., 2012. Gpu Programming in Matlab, MathWorks News&Notes. The MathWorks Inc, Natick, MA, pp. 22–25.

Thömmes, G., Becker, J., Junk, M., Vaikuntam, A.K., Kehrwald, D., Klar, A., Steiner, K., Wiegmann, A., 2009. A lattice boltzmann method for immiscible multiphase flow simulations using the level set method. J. Comput. Phys. 228, 1139–1156.

Tran, N.-P., Lee, M., Choi, D.H., 2015. Memory-efficient parallelization of 3d lattice boltzmann flow solver on a gpu. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). IEEE, pp. 315–324.

Wang, Z., Yan, Z., Chen, G., 2011. Lattice boltzmann method of active contour for image segmentation. In: 2011 Sixth International Conference on Image and Graphics (ICIG). IEEE, pp. 338–343.

Wang, Z., Zhao, Y., Sawchuck, A.P., Dalsing, M.C., Yu, H.W., 2015. Gpu acceleration of volumetric lattice Boltzmann method for patient-specific computational hemodynamics. Comput. Fluids 115, 192–200.

Yu, H., Chen, X., Wang, Z., Deep, D., Lima, E., Zhao, Y., Teague, S.D., 2014. Mass-conserved volumetric lattice boltzmann method for complex flows with willfully moving boundaries. Phys. Rev. E 89, 063304.