Hindawi Publishing Corporation
Scientific Programming
Volume 2017, Article ID 1205892, 16 pages
https://doi.org/10.1155/2017/1205892

Research Article

Performance Optimization of 3D Lattice Boltzmann Flow Solver on a GPU

Nhat-Phuong Tran, Myungho Lee, and Sugwon Hong

Department of Computer Science and Engineering, Myongji University, 116 Myongji-ro, Cheoin-gu, Yongin, Gyeonggi-do, Republic of Korea

Correspondence should be addressed to Myungho Lee; myunghol@mju.ac.kr

Received 9 June 2016; Accepted 23 October 2016; Published 16 January 2017

Academic Editor: Basilio B. Fraguela

Lattice Boltzmann Method (LBM) is a powerful numerical simulation method of the fluid flow. With its data parallel nature, it is a promising candidate for a parallel implementation on a GPU. The LBM, however, is heavily data intensive and memory bound. In particular, moving the data to the adjacent cells in the streaming computation phase incurs a lot of uncoalesced accesses on the GPU which affects the overall performance. Furthermore, the main computation kernels of the LBM use a large number of registers per thread which limits the thread parallelism available at the run time due to the fixed number of registers on the GPU. In this paper, we develop high performance parallelization of the LBM on a GPU by minimizing the overheads associated with the uncoalesced memory accesses while improving the cache locality using the tiling optimization with the data layout change. Furthermore, we aggressively reduce the register uses for the LBM kernels in order to increase the run-time thread parallelism. Experimental results on the Nvidia Tesla K20 GPU show that our approach delivers impressive throughput performance: 1210.63 Million Lattice Updates Per Second (MLUPS).

1. Introduction

Lattice Boltzmann Method (LBM) is a powerful numerical simulation method of the fluid flow, originating from the lattice gas automata methods [1]. LBM models the fluid flow consisting of particles moving with random motions. Such particles exchange the momentum and the energy through the streaming and the collision processes over the discrete lattice grid in the discrete time steps. At each time step, the particles move into adjacent cells which cause collisions with the existing particles in the cells. The intrinsic data parallel nature of LBM makes this class of applications a promising candidate for parallel implementation on various High Performance Computing (HPC) architectures including many-core accelerators such as the Graphic Processing Unit (GPU) [2], Intel Xeon Phi [3], and the IBM Cell BE [4].

Recently, the GPU is becoming increasingly popular for the HPC server market and in the Top 500 list, in particular. The architecture of the GPU has gone through a number of innovative design changes in the last decade. It is integrated with a large number of cores and multiple threads per core, levels of the cache hierarchies, and the large amount (>5 GB) of the on-board memory. The peak floating-point throughput performance (flops) of the latest GPU has drastically increased to surpass 1 Tflops for the double precision arithmetic [5]. In addition to the architectural innovations, user friendly programming environments have been recently developed such as CUDA [5] from Nvidia, OpenCL [6] from Khronos Group, and OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB). The advanced GPU architecture and the flexible programming environments have made possible innovative performance improvements in many application areas.

In this paper, we develop high performance parallelization of the LBM on a GPU. The LBM is heavily data intensive and memory bound. In particular, moving the data to the adjacent cells in the streaming phase of the LBM incurs a lot of uncoalesced accesses on the GPU and affects the overall performance. Previous research focused on utilizing the shared memory of the GPU to deal with the problem [1, 8, 9]. In this paper, we use the tiling algorithm along with the data layout change in order to minimize the overheads

of the uncoalesced accesses and improve the cache locality as well. The computation kernels of the LBM involve a large number of floating-point variables, thus using a large number of registers per thread. This limits the available thread parallelism generated at the run time as the total number of the registers on the GPU is fixed. We developed techniques to aggressively reduce the register uses for the kernels in order to increase the available thread parallelism and the occupancy on the GPU. Furthermore, we developed techniques to remove the branch divergence. Our parallel implementation using CUDA shows impressive performance results. It delivers up to 1210.63 Million Lattice Update Per Second (MLUPS) throughput performance and 136-time speedup on the Nvidia Tesla K20 GPU compared with a serial implementation.

The rest of the paper is organized as follows: Section 2 introduces the LBM algorithm. Section 3 describes the architecture of the latest GPU and its programming model. Section 4 explains our techniques for minimizing the uncoalesced accesses, improving the cache locality and the thread parallelism along with the register usage reduction and the branch divergence removal. Section 5 shows the experimental results on the Nvidia Tesla K20 GPU. Section 6 explains the previous research on paralleling the LBM. Section 7 wraps up the paper with conclusions.

2. Lattice Boltzmann Method

Lattice Boltzmann Method (LBM) is a powerful numerical simulation of the fluid flow. It is derived as a special case of the lattice gas cellular automata (LGCA) to simulate the fluid motion. The fundamental idea is that the fluids can be regarded as consisting of a large number of small particles moving with random motions. These particles exchange the momentum and the energy through the particle streaming and the particle collision. The physical space of the LBM is discretized into a set of uniformly spaced nodes (lattice). At each node, a discrete set of velocities is defined for the propagation of the fluid molecules. The velocities are referred to as microscopic velocities which are denoted by $\overrightarrow{e_i}$. The LBM model which has n dimensions and q velocity vectors at each lattice point is represented as DnQq. Figure 1 shows a typical lattice node of the most common model in 2D (D2Q9) which has two-dimensional 9 velocity vectors. In this paper, however, we consider the D3Q19 model which has three-dimensional 19 velocity vectors. Figure 2 shows a typical lattice node of D3Q19 model with 19 velocities $\overrightarrow{e_i}$ defined by

$$\overrightarrow{e_i} = \begin{cases} (0,0,0), & i = 0 \\ (\pm 1,0,0), (0,\pm 1,0), (0,0,\pm 1) & i = 2,4,6,8,9,14 \\ (\pm 1,\pm 1,0), (0,\pm 1,\pm 1), (\pm 1,0,\pm 1) & i = 1,3,5,7,10,11,12,13,15,16,17,18. \end{cases} \tag{1}$$

(i) Each particle on the lattice is associated with a discrete distribution function, called as particle distribution function (pdf), $f_i(\vec{x},t), i = 0,\ldots,18$. The LB equation is discretized as follows:

$$f_i\left(\vec{x} + c\overrightarrow{e_i}\Delta t, t + \Delta t\right) = f_i(\vec{x},t) - \frac{1}{\tau}\left[f_i(\vec{x},t) - f_i^{(\mathrm{eq})}\left(\rho(\vec{x},t), \vec{u}(\vec{x},t)\right)\right] \tag{2}$$

where c is the lattice speed and τ is the relaxation parameter.

(ii) The macroscopic quantities are the density ρ and the velocity $\vec{u}(\vec{x},t)$. They are defined as

$$\rho(\vec{x},t) = \sum_{i=0}^{18} f_i(\vec{x},t) \tag{3}$$

$$\vec{u}(\vec{x},t) = \frac{1}{\rho}\sum_{i=0}^{18} c f_i \overrightarrow{e_i} \tag{4}$$

(iii) The equilibrium function $f_i^{(\mathrm{eq})}(\rho(\vec{x},t), \vec{u}(\vec{x},t))$ is defined as

$$f_i^{(\mathrm{eq})}\left(\rho(\vec{x},t), \vec{u}(\vec{x},t)\right) = \omega_i \rho + \rho s_i\left(\vec{u}(\vec{x},t)\right) \tag{5}$$

where

$$s_i(\vec{u}) = \omega_i\left[1 + \frac{3}{c^2}\left(\overrightarrow{e_i}\cdot\vec{u}\right) + \frac{9}{2c^4}\left(\overrightarrow{e_i}\cdot\vec{u}\right)^2 - \frac{3}{2c^2}\vec{u}\cdot\vec{u}\right] \tag{6}$$

and the weighting factor ω_i has the following values:

$$\omega_i = \begin{cases} \frac{1}{3}, & \alpha = 0 \\ \frac{1}{18}, & \alpha = 2,4,6,8,9,14 \\ \frac{1}{36}, & \alpha = 1,3,5,7,10,11,12,13,15,16,17,18. \end{cases} \tag{7}$$

Algorithm 1 summarizes the algorithm of the LBM. The LBM algorithm executes a loop over a number of time steps. At each iteration, two computation steps are applied:

(i) Streaming (or propagation) phase: the particles move according to the pdf into the adjacent cells.

(1) Step 1: Initialize macroscopic quantities, density ρ, velocity $\vec{u}$, the distribution function f_i, and the equilibrium function $f_i^{(eq)}$
(2) Step 2: Streaming phase: move $f_i \to f_i^*$ in the direction of $\vec{e_i}$
(3) Step 3: Calculate density ρ and velocity $\vec{u}$ from f_i^* using Equations (3) and (4)
(4) Step 4: Calculate the equilibrium function $f_i^{(eq)}$ using Equation (5)
(5) Step 5: Collision phase: calculate the updated distribution function $f_i = f_i^* - (1/\tau)(f_i^* - f_i^{eq})$ using Equation (2)
(6) Repeat Steps 2 to 5 *timeSteps*-times

Algorithm 1: Algorithm of LBM.

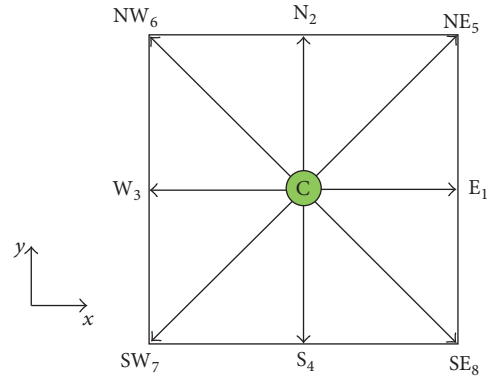


Figure 1: Lattice cell with 9 discrete directions in D2Q9 model.

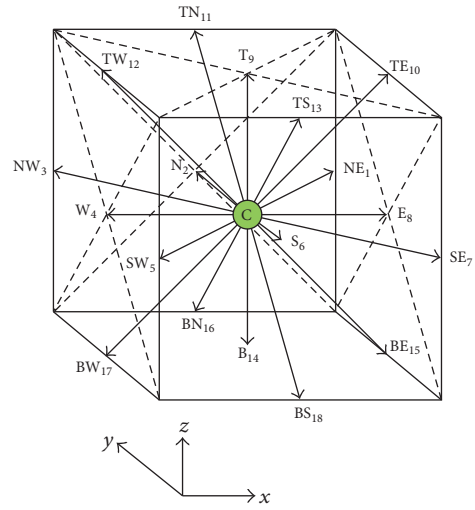


Figure 2: Lattice cell with 19 discrete directions in D3Q19 model.

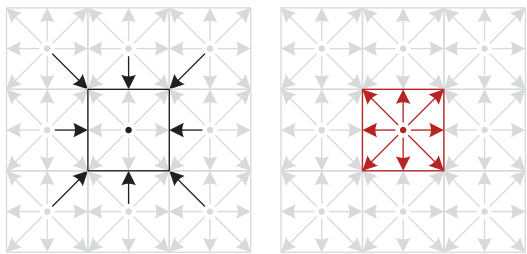
Table 1: Pull and push schemes.

Pull scheme	Push scheme
+ Read distribution functions from the adjacent cells $f_i\left(\vec{x_i} - c\vec{e_i}\Delta t, t - \Delta t\right)$	+ Read distribution functions from the current cell $f_i(\vec{x_i}, t)$
+ Calculate $\rho, \vec{u}, f_i^{(eq)}$	+ Calculate $\rho, \vec{u}, f_i^{(eq)}$
+ Update values to the current cell $f_i(\vec{x_i}, t)$	+ Update values to the adjacent cells $f_i(\vec{x_i} + c\vec{e_i}\Delta t, t + \Delta t)$

(ii) Collision phase: the particles collide with other particles streaming into this cell from different directions.

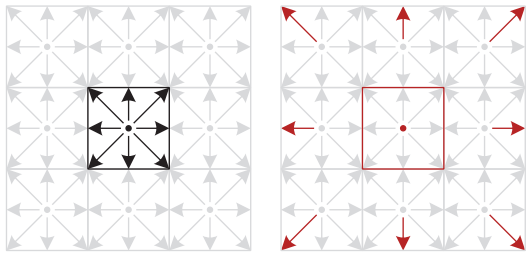
Depending on whether the streaming phase precedes or follows the collision phase, we have the pull or the push scheme in the update process [10]. The pull scheme (Figure 3) pulls the post-collision values from the previous time step from lattice A and then performs the collision on these to produce the new pdfs which are stored in lattice B. In the push scheme (Figure 4); on the other hand, the pdfs of one node (square with black arrows) are read from lattice A; then collision step performs first. The post-collision values are propagated to the neighbor nodes in the streaming step to lattice B (red arrows). Table 1 compares the computation steps of these schemes.

In Algorithm 2, we list the skeleton of the LBM algorithm which consists of the collision phase and the streaming phase. In the function *LBM*, the *collide* function and *stream* function are called *timeSteps*-times. At the end of each time step,



(a) Pull: lattice A (b) Pull: lattice B

Figure 3: Illustration of pull scheme with D2Q9 model [11].



(a) Push: lattice A (b) Push: lattice B

Figure 4: Illustration of push scheme with D2Q9 model [11].

```
(1)  void LBM(double *source_grid, double
        *dest_grid, int grid_size, int timeSteps)
(2)  {
(3)    int i;
(4)    double *temp_grid;
(5)    for (i = 0; i < timeSteps; i++)
(6)    {
(7)      collide(source_grid, temp_grid, grid_size);
(8)      stream(temp_grid, dest_grid, grid_size);
(9)      swap_grid(source_grid, dest_grid);
(10)   }
(11) }
```

Algorithm 2: Basic skeleton of LBM algorithm.

source_grid and *dest_grid* are swapped to interchange values between the two grids.

3. Latest GPU Architecture

Recently, the many-core accelerator chips are becoming increasingly popular for the HPC applications. The GPU chips from Nvidia and AMD are representative ones along with the Intel Xeon Phi. The latest GPU architecture is characterized by a large number of uniform fine-grain programmable cores or thread processors which have replaced separate processing units for shader, vertex, and pixel in the earlier GPUs. Also, the clock rate of the latest GPU has ramped up significantly. These have drastically improved the floating-point performance of the GPUs, far exceeding that of the latest CPUs. The fine-grain cores (or thread processors) are distributed in multiple streaming multiprocessors (SMX) (or thread blocks) (see Figure 5). Software threads are divided into a number of thread groups (called WARPs) each of which consists of 32 threads. Threads in the same WARP are scheduled and executed together on the thread processors in the same SMX in the SIMD (Single Instruction Multiple Data) mode. Each thread executes the same instruction directed by the common Instruction Unit on its own data streaming from the device memory to the on-chip cache memories and registers. When a running WARP encounters a cache miss, for example, the context is switched to a new WARP while the cache miss is serviced for the next few hundred cycles, the GPU executes in a multithreaded fashion as well.

The GPU is built around a sophisticated memory hierarchy as shown in Figure 5. There are registers and local memories belonging to each thread processor or core. The local memory is an area in the off-chip device memory. Shared memory, level-1 (L-1) cache, and read-only data cache are integrated in a thread block of the GPU. The shared memory is a fast (as fast as registers) programmer-managed memory. Level-2 (L-2) cache is integrated on-chip and used among all the thread blocks. Global memory is an area in the off-chip device memory accessed from all the thread blocks, through which the GPU can communicate with the host CPU. Data in the global memory get cached directly in the shared memory by the programmer or they can be cached through

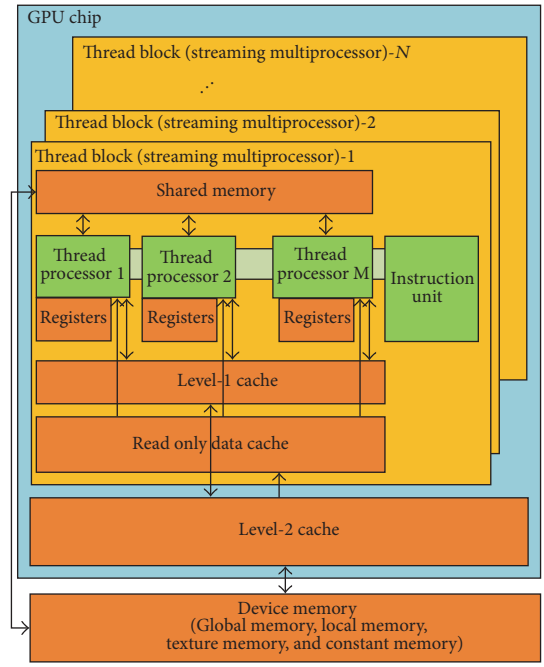


Figure 5: Architecture of a latest GPU (Nvidia Tesla K20).

the L-2 and L-1 caches automatically as they get accessed. There are constant memory and texture memory regions in the device memory also. Data in these regions is read-only. They can be cached in the L-2 cache and the read-only data cache. On Nvidia Tesla K20, the read-only data from the global memory can be loaded through the same cache used by the texture pipeline via a standard pointer without the need to bind to a texture beforehand. This read-only cache is used automatically by the compiler as long as certain conditions are met. __restrict__ qualifier should be used when a variable is declared to help the compiler detect the conditions [5].

In order to efficiently utilize the latest advanced GPU architectures, programming environments such as CUDA [5] from Nvidia, OpenCL [6] from Khronos Group, and OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB) have been developed. Using these environments, users can have a more direct control over the

```
(1) int i;
(2) for(i = 0; i < timeSteps; i++)
(3) {
(4)   collision_kernel<<<GRID, BLOCK>>>
          (source_grid, temp_grid, xdim, ydim,
          zdim, cell_size, grid_size);
(5)   cudaThreadSynchronize();
(6)   streaming_kernel<<<GRID, BLOCK>>>(temp_grid,
          dest_grid, xdim, ydim, zdim, cell_size,
          grid_size);
(7)   cudaThreadSynchronize();
(8)   swap_grid(source_grid, dest_grid);
(9) }
```

Algorithm 3: Two separate CUDA kernels for different phases of LBM.

large number of GPU cores and its sophisticated memory hierarchy. The flexible architecture and the programming environments have led to a number of innovative performance improvements in many application areas and many more are still to come.

4. Optimizing Cache Locality and Thread Parallelism

In this section, we first introduce some preliminary steps we employed in our parallelization and optimization of the LBM algorithm in Section 4.1. They are mostly borrowed from the previous research such as combining the collision phase and the streaming phase, a GPU architecture friendly data organization scheme (SoA scheme), an efficient data placement in the GPU memory hierarchy, and using the pull scheme for avoiding and minimizing the uncoalesced memory accesses. Then, we describe our key optimization techniques for improving the cache locality and the thread parallelism such as the tiling with the data layout change and the aggressive reduction of the register uses per thread in Sections 4.2 and 4.3. Optimization techniques for removing the branch divergence are presented in Section 4.4. Our key optimization techniques presented in this section have been improved from our earlier work in [13].

4.1. Preliminaries

4.1.1. Combination of Collision Phase and Streaming Phase. As shown in the description of the LBM algorithm in Algorithm 2, the LBM consists of the two main computing kernels: *collision_kernel* for the collision phase and *streaming_kernel* for the streaming phase. In the *collision_kernel*, threads load the particle distribution function from the source grid (*source_grid*) and then calculate the velocity, the density, and the collision product. The post-collision values are stored to the temporary grid (*temp_grid*). In *streaming_kernel*, the post-collision values from *temp_grid* are loaded and updated to appropriate neighbor grid cells in the destination grid (*dest_grid*). At the end of each time step, *source_grid* and *dest_grid* are swapped for the next time step. This implementation (see Algorithm 3) needs extra loads/stores from/to *temp_grid* which is stored in the global memory and affects the global memory bandwidth [2]. In addition, some extra cost is incurred with the global synchronization between the two kernels (*cudaThreadSynchronize*) which affects the overall performance. In order to reduce these overheads, we can combine *collision_kernel* and *streaming_kernel* into one kernel *lbm_kernel*, where the collision product is streamed to the neighbor grid cells directly after calculation (see Algorithm 4). Compared with Algorithm 3, storing to and loading from *temp_grid* are removed and the global synchronization cost is reduced.

```
(1) int i;
(2) for (i = 0; i < timeSteps; i++)
(3) {
(4) lbm_kernel<<<GRID, BLOCK>>>(source_grid,
       dest_grid, xdim, ydim, zdim, cell_size,
       grid_size);
(5) cudaThreadSyncronize();
(6) swap_grid(source_grid, dest_grid);
(7) }
```

Algorithm 4: Single CUDA kernel after combining two phases of LBM.

4.1.2. Data Organization. In order to represent the 3-dimensional grid of cells, we use the 1-dimensional array which has $N_x \times N_y \times N_z \times Q$ elements, where N_x, N_y, N_z are width, height, and depth of the grid and Q is the number of directions of each cell [2, 14]. For example, if the model is $D3Q19$ with $N_x = 16$, $N_y = 16$, and $N_z = 16$, we have the 1D array of $16 \times 16 \times 16 \times 19 = 77824$ elements. We use 2 separate arrays for storing the source grid and the destination grid.

There are two common data organization schemes for storing the arrays:

(i) Array of structures (AoS): grid cells are arranged in 1D array. 19 distributions of each cell occupy 19 consecutive elements of the 1D array (Figure 6).

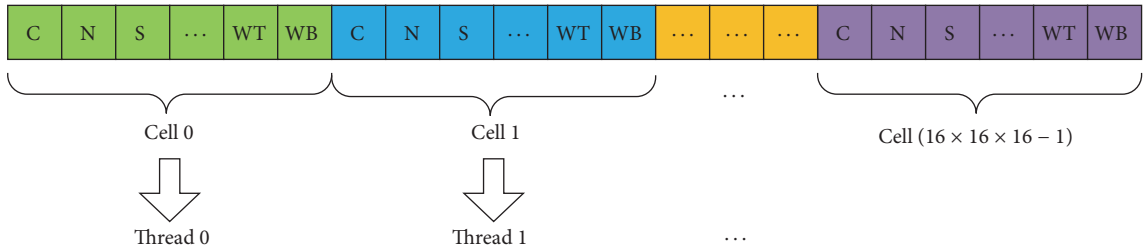


Figure 6: AoS scheme.

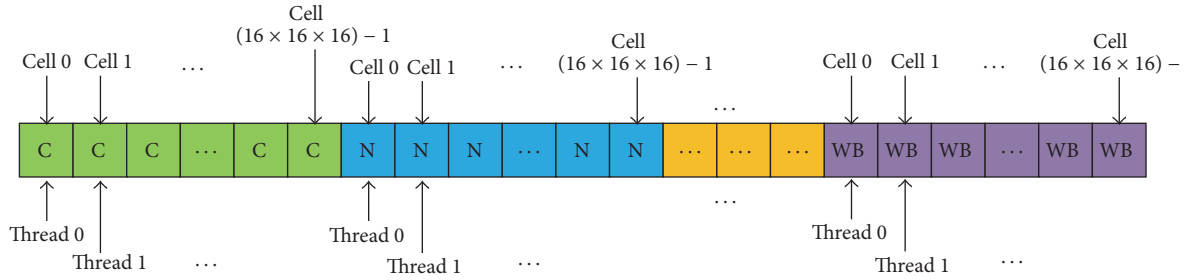


Figure 7: SoA scheme.

(ii) Structure of arrays (SoA): the value of one distribution of all cells is arranged consecutively in memory (Figure 7). This scheme is more suitable for the GPU architecture as we will show in the experimental results (Section 5).

4.1.3. Data Placement. In order to efficiently utilize the memory hierarchy of the GPU, the placement of the major data structures of the LBM is crucial [5]. In our implementation, we use the following arrays: *src_grid*, *dst_grid*, *types_arr*, *lc_arr*, and *nb_arr*. We use the following data placements for these arrays:

(i) *src_grid* and *dst_grid* are used to store the input grid and the result grid. They are swapped at the end of each time step by exchanging their pointers instead of explicit storing to and loading from the memory through a temporary array. Since *src_grid* and *dst_grid* are very large size arrays with a lot of data stores and loads, we place them in the global memory.

(ii) In *types_arr* array, the types of the grid cells are stored. We use the Lid Driven Cavity (LDC) as the test case in this paper. The LDC consists of a cube filled with the fluid. One side of the cube serves as the acceleration plane by sliding constantly. The acceleration is implemented by assigning the cells in the acceleration area at a constant velocity. This change requires three types of cells: regular fluid, acceleration cells, or boundary. Thus, we also need 1D array, *types_arr*, in order to store the types of each cell in the grid. The size of this array is $N_x \times N_y \times N_z$ elements. For example, if the model is D3Q19 with $N_x = 16$, $N_y = 16$, and $N_z = 16$, the size of the array is $16 \times 16 \times 16 = 4{,}096$ elements. Thus, *types_arr* is a large array also and contains constant values. Thus, they are not modified throughout the execution of the program. For these reasons, the texture memory is the right place for this array.

(iii) *lc_arr* and *nb_arr* are used to store the base indices for accesses to 19 directions of the current cell and the neighbor cells, respectively. There are 19 indices corresponding to 19 directions of D3Q19 model. These indices are calculated at the start of the program and used till the end of the program execution. Thus, we use the constant memory to store them. As standing at any cell, we use the following formula to define the position in the 1D array of any direction out of 19 cell directions: $curr\_dir\_pos\_in\_arr = cell\_pos + lc\_arr[direction]$ (for the current cell) and $nb\_dir\_pos\_in\_arr = nb\_pos + nb\_arr[direction]$ (for the neighbor cells).

4.1.4. Using Pull Scheme to Reduce Costs for Uncoalesced Accesses. Coalescing the global memory accesses can significantly reduce the memory overheads on the GPU. Multiple global memory loads whose addresses fall within the 128-byte range are combined into one request and sent to the memory. This saves the memory bandwidth a lot and improves the performance. In order to reduce the costs for the uncoalesced accesses, we use the pull scheme [12]. Choosing the pull scheme comes from the observation that the cost of the uncoalesced reading is smaller than the cost of the uncoalesced writing.

Algorithm 5 shows the LBM algorithm using the push scheme. At the first step, the pdfs are copied directly from the current cell. These pdfs are used to calculate the pdfs at the new time step (collision phase). The new pdfs are then streamed to the adjacent cells (streaming phase). At

```
(1) __global__ void soa_push_kernel(float
     *source_grid, float *dest_grid, unsigned
     char* flags)
(2) {
(3)   Gather 19 pdfs from the current cell
(4)
(5)   Apply boundary conditions
(6)
(7)   Calculate the mass density ρ and the velocity u
(8)
(9)   Calculate the local equilibrium distribution
        functions f^(eq) using ρ and u
(10)
(11)  Calculate the pdfs at new time step
(12)
(13)  Stream 19 pdfs to the adjacent cells
(14) }
```

Algorithm 5: Kernel using the *push* scheme.

```
(1) __global__ void soa_pull_kernel(float
     *source_grid, float *dest_grid, unsigned
     char* flags)
(2) {
(3)   Stream 19 pdfs from adjacent cells to the
        current cell
(4)
(5)   Apply boundary conditions
(6)
(7)   Calculate the mass density ρ and the velocity u
(8)
(9)   Calculate the local equilibrium distribution
        functions f^(eq) using ρ and u
(10)
(11)
(12)  Calculate the pdf at new time step
(13)
(14)  Save 19 values of pdf to the current cell
(15) }
```

Algorithm 6: Kernel using the *pull* scheme.

the streaming phase, the distribution values are updated to neighbors after they are calculated. All distribution values which do not move to the east or west direction (x-direction values equal to 0) can be updated to the neighbors (write to the device memory) directly without any misalignment. However, other distribution values (x-direction values equal to +1 or −1) need to be considered carefully because of their misaligned update positions. The update positions are shifted to the memory locations that do not belong to the 128-byte segment while thread indexes are not shifted correspondingly. So the misaligned accesses occur and the performance can degrade significantly.

If we use the pull scheme, on the other hand, the order of the collision phase and the streaming phase in the LBM kernel is reversed (see Algorithm 6). At the first step of the pull scheme, the pdfs from adjacent cells are gathered to the current cell (streaming phase) (Lines 3–5). Next, these pdfs are used to calculate the pdfs at the new time step and these new pdfs are then stored to the current cell directly (collision phase). Thus, in the pull scheme, the uncoalesced accesses occur when the data is read from the device memory whereas they occur when the data is written in the push scheme. As a result, the cost of the uncoalesced accesses is smaller with the pull scheme.

4.2. Tiling Optimization with Data Layout Change. In the D3Q19 model of the LBM, as computations for the streaming and the collision phases are conducted for a certain cell, 19 distribution values which belong to 19 surrounding cells are accessed. Figure 8 shows the data accesses to the 19 cells when

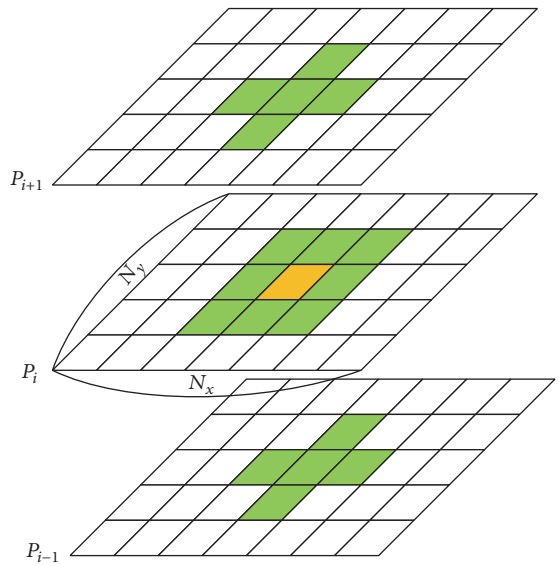


FIGURE 8: Data accesses for orange cell in conducting computations for streaming and collision phases.

a thread performs the computations for the orange colored cell. The 19 cells (18 directions (green cells) + current cell in the center (orange cell)) are distributed on the three different planes. Let P_i be the plane containing the current computing (orange) cell, and let P_{i-1} and P_{i+1} be the lower and upper planes, respectively. P_i plane contains 9 cells. P_{i-1} and P_{i+1} planes contain 5 cells, respectively. When the computations for the cell, for example, $(x, y, z) = (1, 1, 1)$, are performed, the following cells are accessed:

(i) P_0 plane: (0, 1, 0), (1, 0, 0), (1, 1, 0), (1, 2, 0), (2, 1, 0)

(ii) P_1 plane: (0, 0, 1), (0, 1, 1), (0, 2, 1), (1, 0, 1), (1, 1, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 2, 1)

(iii) P_2 plane: (0, 1, 2), (1, 0, 2), (1, 1, 2), (1, 2, 2), (2, 1, 2)

The 9 accesses for P_1 plane are divided into three groups {(0,0,1), (0,1,1), (0,2,1)}, {(1,0,1), (1,1,1), (1,2,1)}, {(2,0,1), (2,1,1), (2,2,1)}. Each group accesses the consecutive memory locations belonging to the same row. Accesses of the different groups are separated apart and lead to the uncoalesced accesses on the GPU when N_x is sufficiently large. In each of P_0 and P_2 planes, there are three groups of accesses each. Here, the accesses of the same group touch the consecutive memory locations and accesses of the different groups are separated apart in the memory which lead to the uncoalesced accesses also. Accesses to the data elements in the different planes (P_0, P_1, and P_2) are further separated apart and also lead to the uncoalesced accesses when N_y is sufficiently large.

As the computations proceed, three rows in the y-dimension of P_0, P_1, P_2 planes will be accessed sequentially for $x = 0 \sim N_x - 1$, $y = 0, 1, 2$, followed by $x = 0 \sim N_x - 1$, $y = 1, 2, 3, \ldots$, $x = 0 \sim N_x - 1$, $y = N_y - 3, N_y - 2, N_y - 1$. When the complete P_0, P_1, P_2 planes are swept, then similar data accesses will continue for P_1, P_2, and P_3 planes, and so on. Therefore, there are a lot of data reuses in x-, y-, and z-dimensions. As explained in Section 4.1.2, the 3D lattice grid is stored in the 1D array. The 19 cells for the computations belonging to the same plane are stored ± 1 or $\pm N_x + \pm 1$ cells away. The cells in different planes are stored $\pm N_x \times Ny + \pm N_x + \pm 1$ cells away. The data reuse distance along the x-dimension is short: +1 or +2 loop iterations apart. The data reuse distance along the y- and z-dimensions is $\pm N_x + \pm 1$ or $\pm N_x \times N_y + \pm N_x + \pm 1$ iterations apart. If we can make the data reuse occur faster by reducing the reuse distances, for example, using the tiling optimization, it can greatly improve the cache hit ratio. Furthermore, it can reduce the overheads with the uncoalesced accesses because lots of global memory accesses can be removed by the cache hits. Therefore, we tile the 3D lattice grid into smaller 3D blocks. We also change the data layout in accordance with the data access patterns of the tiled code in order to store the data elements in different groups closer in the memory. Thus we can remove a lot of uncoalesced memory accesses, because they can be stored within 128-byte boundary. In Sections 4.2.1 and 4.2.2, we describe our tiling and data layout change optimizations.

4.2.1. Tiling. Let us assume the following:

(i) N_x, N_y, and N_z are sizes of the grid in x-, y-, and z-dimension.

(ii) n_x, n_y, and n_z are sizes of the 3D block in x-, y-, and z-dimension.

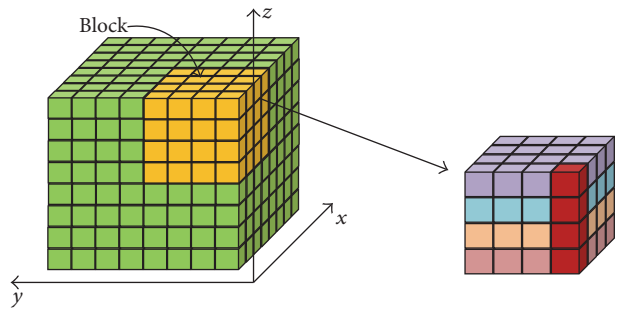
(iii) xy-plane is a subplane which is composed of $(n_x \times n_y)$ cells.

We tile the grid into small 3D blocks with the tile sizes of n_x, n_y, and n_z (yellow block in Figure 9(a)), where

$$\begin{aligned} n_x &= N_x \div x_c \\ n_y &= N_y \div y_c \\ n_z &= N_z \div z_c \\ x_c, y_c, z_c &= [1, 2, 3, \ldots). \end{aligned} \tag{8}$$

We let each CUDA thread block process one 3D tiled block. Thus n_z xy-planes need to be loaded for each thread block. In each xy-plane, each thread of the thread block executes the computations for one grid cell. Thus each thread deals with a column containing n_z cells (the red column in Figure 9(b)). If $z_c = 1$, each thread processes N_z cells and if $z_c = N_z$, each thread processes only one cell. The tile size can be adjusted by changing the constants x_c, y_c, and z_c. These constants need to be selected carefully to optimize the performance. Using the tiling, the number of created threads is reduced by z_c-times.

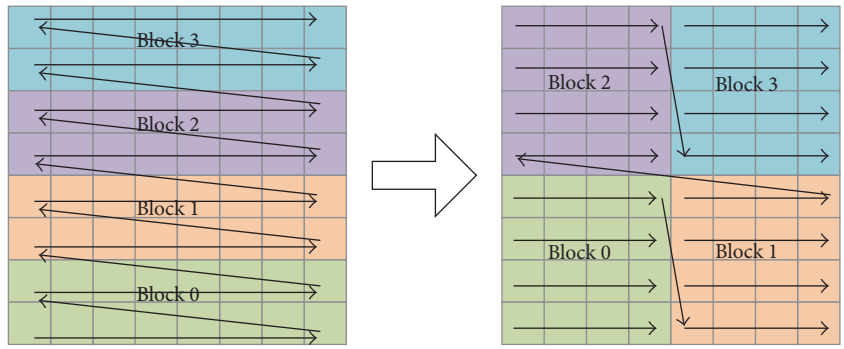
4.2.2. Data Layout Change. In order to further improve benefits of the tiling and reduce the overheads associated with the uncoalesced accesses, we propose to change the data layout. Figure 10 shows one xy-plane of the grid with and without the layout change. With the original layout (Figure 10(a)), the data is stored in the row major fashion. Thus the entire first row is stored, followed by the second



(a) Grid divided into 3D blocks

(b) Block containing subplanes. The red column contains all cells one thread processes

Figure 9: Tiling optimization for LBM.



(a) Original data layout

(b) Proposed data layout

Figure 10: Different data layouts for blocks.

row, and so on. In the proposed new layout, the cells in the tiled first row in Block 0 are stored first. Then the second tiled row of Block 0 is stored instead of the first row of Block 1 (Figure 10(b)). With the layout change, the data cells accessed in the consecutive iterations of the tiled code are placed sequentially. This places the data elements of the different groups closer. Thus, it increases the possibility for these memory accesses to the different groups coalesced if the tiling factor and the memory layout factor are adjusted appropriately. This can further improve the performance beyond the tiling.

The data layout can be transformed using the following formula:

$$\text{index}_{\text{new}} = x_{\text{id}} + y_{\text{id}} \times N_x + z_{\text{id}} \times N_x \times N_y \quad (9)$$

where x_{id} and y_{id} are cell indexes in x- and y-dimension on the plane of grid and z_{id} is the value in the range of 0 to $n_z - 1$.

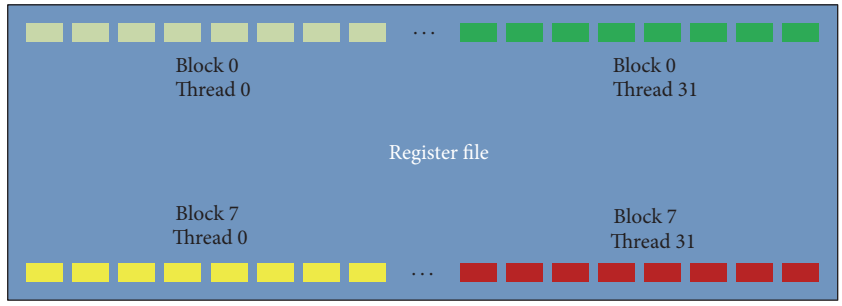
x_{id} and y_{id} can be calculated as follows:

$$\begin{aligned} x_{\text{id}} &= (\text{block index in } x\text{-dimension}) \\ &\quad \times (\text{number of threads in thread block in } x\text{-dimension}) \\ &\quad + (\text{thread index in thread block in } x\text{-dimension}) \\ y_{\text{id}} &= (\text{block index in } y\text{-dimension}) \\ &\quad \times (\text{number of threads in thread block in } y\text{-dimension}) \\ &\quad + (\text{thread index in thread block in } y\text{-dimension}) \end{aligned} \quad (10)$$

In our implementation, we use the changed input data layout stored offline before the program starts. (The original input is changed to the new layout and stored to the input file.) Then, the input file is used while conducting the experiments.

4.3. Reduction of Register Uses per Thread. The D3Q19 model is more precise than the models with smaller distributions such as D2Q9 or D3Q13, thus using more variables. This leads to more register uses for the main computation kernels. In GPU, the register use of the threads is one of the factors limiting the number of active WARPs on a streaming multiprocessor (SMX). Higher register uses can lead to the lower parallelism and occupancy (see Figure 11 for an example) which results in the overall performance degradation. The Nvidia compiler provides a flag to limit the register uses to a certain limit such as *–maxrregcount* or __launch_bounds__() qualifier [5]. The *–maxrregcount* switch sets a maximum on the number of registers used for each thread. These can help increase the occupancy by reducing the register uses per thread. However, our experiments show that the overall performance goes down, because they lead to a lot of register spills/refills to/from the local memory. The increased

(a) Eight blocks, 64 threads per block, and 4 registers per thread



(b) Eight blocks, 32 threads per block, and 8 registers per thread

Figure 11: Sharing 2048 registers among (a) a larger number of threads with smaller register uses versus (b) a smaller number of threads with larger register uses.

memory traffic to/from the local memory and the increased instruction count for accessing the local memory hurt the performance.

In order to reduce the register uses per thread while avoiding the register spill/refill to/from the local memory, we used the following techniques:

(i) Calculate indexing address of distributions manually. Each cell has 19 distributions; thus we need 38 variables for storing indexes (19 distributions × 2 memory accesses for load and store) in the D3Q19 model. However, each index variable is used only one time at each execution phase. Thus, we can use only two variables instead of 38, one for calculating the loading indexes and one for calculating the storing indexes.

(ii) Use the shared memory for the commonly used variables among threads, for example, to store the base addresses.

(iii) Casting multiple small size variables into one large variable: for example, we combined 4 char type variables into one integer variable.

(iv) For simple operations which can be easily calculated, we do not store them to the memory variables. Instead, we recompute them later.

(v) We use only one array to store the distributions instead of using 19 arrays separately.

(vi) In the original LBM code, a lot of variables are declared to store the FP computation results which increase the register uses. In order to reduce the register uses, we attempt to reuse variables whose life-time ended earlier in the former code. This may lower the instruction-level parallelism of the kernel. However, it helps increase the thread-level parallelism as more threads can be active at the same time with the reduced register uses per thread.

(vii) In the original LBM code, there are some complicated floating-point (FP) intensive computations used in a number of nearby statements. We aggressively extract these computations as the common subexpressions. It frees the registers involved in the common subexpressions, thus reducing the register uses. It also reduces the number of dynamic instruction counts.

Applying the above techniques in our implementation, the number of registers in each kernel is greatly reduced from 70 registers to 40 registers. It leads to the higher occupancy for the SMXs and the significant performance improvements.

4.4. Removing Branch Divergence. Flow control instructions on the GPU cause the threads of the same WARP to diverge. Thus, the resulting different execution paths get serialized. This can significantly affect the performance of the application program on the GPU. Thus, the branch divergence should be avoided as much as possible. In the LBM code, there are two main problems which can cause the branch divergence:

(i) Solving the streaming at the boundary positions

(ii) Defining actions for corresponding cell types

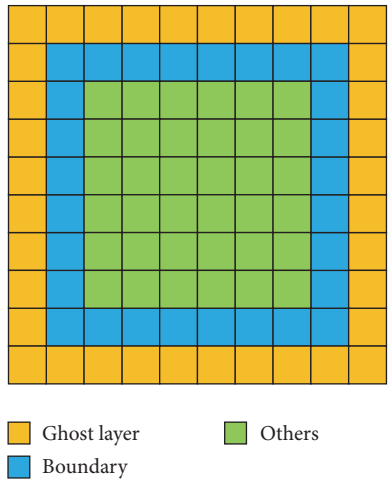


Figure 12: Illustration of the lattice with a "ghost layer."

```
(1) IF(cell_type == FLUID)
(2)   x = a;
(3) ELSE
(4)   x = b;
```

Algorithm 7: Skeleton of *IF-statement* used in LBM kernel.

```
(1) is_fluid = (cell_type == FLUID);
(2) x = a * is_fluid + b * (!is_fluid);
```

Algorithm 8: Code with *IF-statement* removed.

In order to avoid using *IF-statements* while streaming at the boundary position, a "ghost layer" is attached in y- and z-dimension. If N_x, N_y, and N_z are the width, height, and depth of the original grid, $NN_x = N_x$, $NN_y = N_y + 1$, and $NN_z = N_z + 1$ are the new width, height, and depth of the grid with the ghost layer (Figure 12). With the ghost layer, we can regard the computations at the boundary position as the normal ones without worrying about running out of the index bound.

As explained in Section 4.1.2, cells of the grid belong to three types such as the regular fluid, the acceleration cells, or the boundary. The LBM kernel contains conditions to define actions for each type of the cell. The boundary cell type can be covered in the above-mentioned way using the ghost layer. This leads to the existence of the other two different conditions in the same half WARP of the GPU. Thus, in order to remove *IF-conditions* we combine conditions into computational statements. Using this technique, the *IF-statement* in Algorithm 7 is rewritten as in Algorithm 8.

5. Experimental Results

In this section, we first describe the experimental setup. Then we show the performance results with analyses.

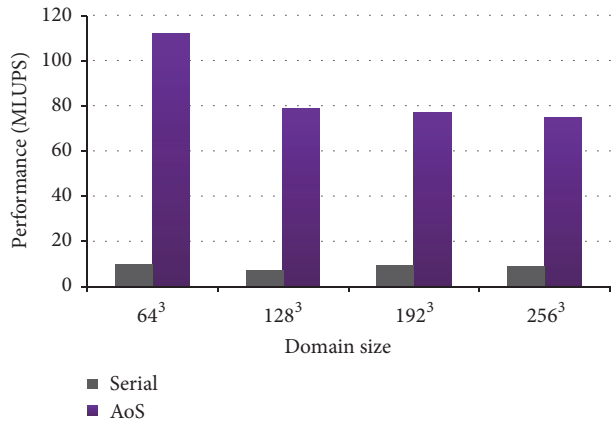


Figure 13: Performance (MLUPS) comparison of the serial and the AoS with different domain sizes.

5.1. Experimental Setup. We implemented the LBM in the following five ways:

(i) Serial implementation using single CPU core (serial), using the source code from the SPEC CPU 2006 470.lbm [15] to make sure it is reasonably optimized

(ii) Parallel implementation on a GPU using the AoS data scheme (AoS)

(iii) Parallel implementation using the SoA data scheme and the push scheme (SoA_Push_Only)

(iv) Parallel implementation using the SoA data scheme and the pull scheme (SoA_Pull_Only)

(v) SoA using pull scheme with our various optimizations including the tiling with the data layout change (SoA_Pull_∗)

We summarize our implementations in Table 2. We used the D3Q19 model for the LBM algorithm. Domain grid sizes are scaled in the range of 64^3, 128^3, 192^3, and 256^3. The numbers of time steps are 1000, 5000, and 10000.

In order to measure the performance of the LBM, the Million Lattice Updates Per Second (MLUPS) unit is used which is calculated as follows:

$$\text{MLUPS} = \frac{N_x \times N_y \times N_z \times N_{\text{ts}}}{10^6 \times T} \tag{11}$$

where N_x, N_y, and N_z are domain sizes in the x-, y-, and z-dimension, N_{ts} is the number of time steps used, and T is the run time of the simulation.

Our experiments were conducted on a system incorporating the Intel multicore processor (6-core 2.0 Ghz Intel Xeon E5-2650) with 20 MB level-3 cache and Nvidia Tesla K20 GPU based on the Kepler architecture with 5 GB device memory. The OS is CentOS 5.5. In order to validate the effectiveness of our approach over the previous approaches, we have also conducted further experiments on another GPU, Nvidia GTX285 GPU.

5.2. Results Using Previous Approaches. The average performances of the serial and the AoS are shown in Figure 13.

Table 2: Summary of experiments.

Experiment	Description
Serial	Serial implementation on single CPU core
Parallel	
AoS	AoS scheme
SoA_Push_Only	SoA scheme + push data scheme
SoA_Pull	
SoA_Pull_Only	SoA scheme + pull data scheme
SoA_Pull_BR	SoA scheme + pull data scheme + branch divergence removal
SoA_Pull_RR	SoA scheme + pull data scheme + register reduction
SoA_Pull_Full	SoA scheme + pull data scheme + branch divergence removal + register usage reduction
SoA_Pull_Full_Tiling	SoA scheme + pull data scheme + branch divergence removal + register usage reduction + tiling with data layout change

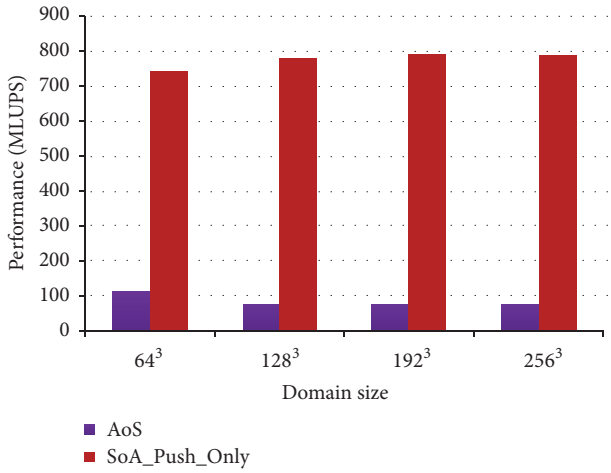


Figure 14: Performance (MLUPS) comparison of the AoS and the SoA with different domain sizes.

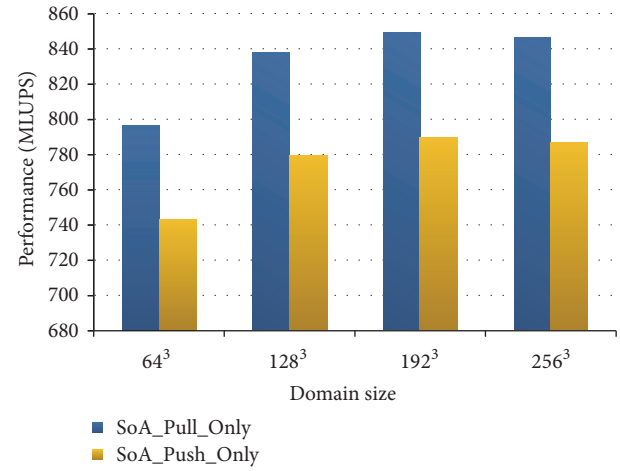


Figure 15: Performance (MLUPS) comparison of the SoA using *push* scheme and *pull* scheme with different domain sizes.

With various domain sizes of 64^3, 128^3, 192^3, and 256^3, the MLUPS numbers for the serial are 9.82 MLUPS, 7.42 MLUPS, 9.57 MLUPS, and 8.93 MLUPS. The MLUPS for the AoS are 112.06 MLUPS, 78.69 MLUPS, 76.86 MLUPS, and 74.99 MLUPS, respectively. With these numbers as the baseline, we also measured the SoA performance for various domain sizes. Figure 14 shows that the SoA significantly outperforms the AoS scheme. The SoA is faster than the AoS by 6.63, 9.91, 10.28, and 10.49 times for the domain sizes 64^3, 128^3, 192^3, and 256^3. Note that in this experiment we applied only the SoA scheme without any other optimization techniques.

Figure 15 compares the performance of the pull scheme and the push scheme. For fair comparison, we did not apply any other optimization techniques to both of the implementations. The pull scheme performs at 797.3 MLUPS, 838.4 MLUPS, 849.8 MLUPS, and 848.37 MLUPS, whereas the push scheme performs at 743.4 MLUPS, 780 MLUPS, 790.16 MLUPS, and 787.13 MLUPS for domain sizes 64^3, 128^3, 192^3, and 256^3, respectively. Thus, the pull scheme is better than the push scheme by 6.75%, 6.97%, 7.02%, and 7.2%, respectively. The number of global memory transactions observed shows that the total transactions (loads and stores) of the pull and push schemes are quite equivalent. However, the number of store transactions of the pull scheme is 56.2% smaller than the push scheme. This leads to the performance improvement of the pull scheme compared with the push scheme.

5.3. Results Using Our Optimization Techniques. In this subsection, we show the performance improvements of our optimization techniques compared with the previous approach based on the SoA_Pull implementation:

(i) Figure 16 compares the average performance of the SoA with and without removing the branch divergences explained in Section 4.4 in the kernel code. Removing the branch divergence improves the performance by 4.37%, 4.45%, 4.69%, and 5.19% for domain sizes 64^3, 128^3, 192^3, and 256^3, respectively.

(ii) Reducing the register uses described in Section 4.3 improves the performance by 12.07%, 12.44%, 11.98%, and 12.58% as Figure 17 shows.

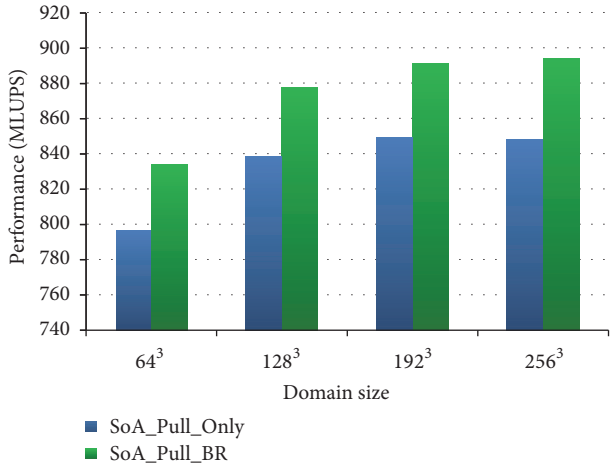


Figure 16: Performance (MLUPS) comparison of the SoA with and without branch removal for different domain sizes.

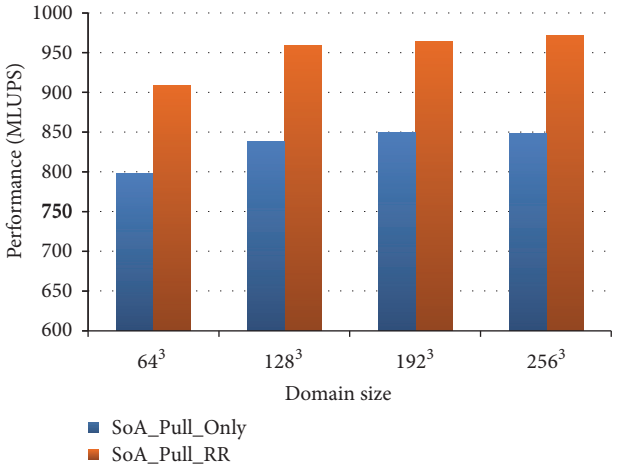


Figure 17: Performance (MLUPS) comparison of the SoA with and without reducing register uses for different domain sizes.

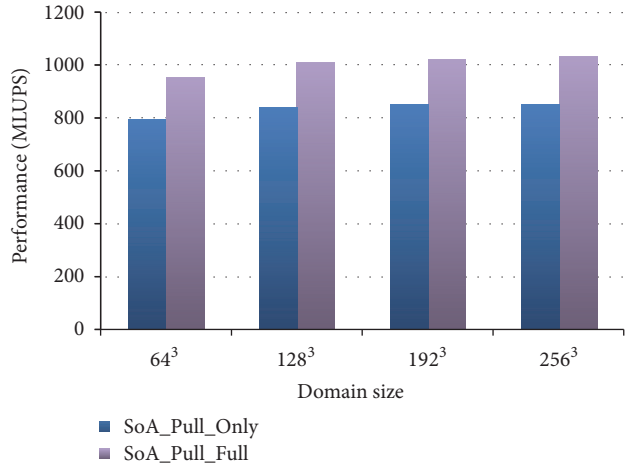


Figure 18: Performance (MLUPS) comparison of the SoA with and without optimization techniques for different domain sizes.

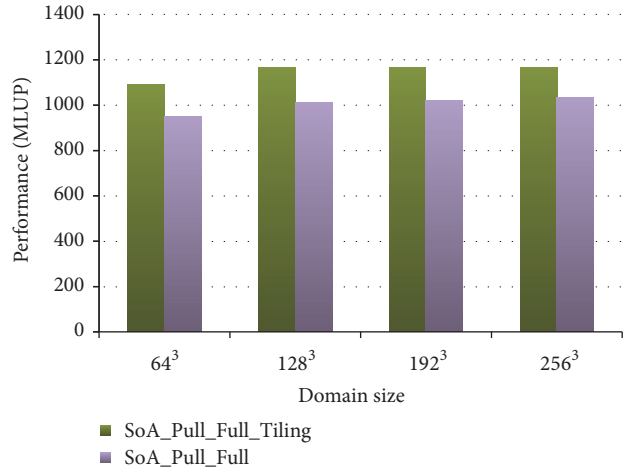


Figure 19: Performance (MLUPS) comparison of the SoA_Pull_Full and the SoA_Pull_Full_Tiling with different domain sizes.

(iii) Figure 18 compares the performance of the SoA using the pull scheme with optimization techniques such as the branch divergence removal and the register usage reduction described in Sections 4.3 and 4.4 (SoA_Pull_Full) and the SoA_Pull_Only. The optimized SoA_Pull implementation is better than the SoA_Pull_Only by 16.44%, 16.89%, 16.68%, and 17.77% for the domain sizes 64^3, 128^3, 192^3, and 256^3, respectively.

(iv) Figure 19 shows the performance comparison of the SoA_Pull_Full and SoA_Pull_Full_Tiling. The SoA_Pull_Full_Tiling performance is better than the SoA_Pull_Full from 11.78% to 13.6%. The domain size 128^3 gives the best performance improvement of 13.6%, while the domain size 256^3 gives the lowest improvement of 11.78%. The experimental results show that the tiling size for the best performance is $n_x = 32$, $n_y = 16$, and $n_z = N_z \div 4$.

(v) Figure 20 presents the overall performance of the SoA_Pull_Full_Tiling implementation compared with the SoA_Pull_Only. With all our optimization techniques described in Sections 4.2, 4.3, and 4.4, we obtained 28% overall performance improvements compared with the previous approach.

(vi) Table 3 compares the performance of four implementations (serial, AoS, SoA_Pull_Only, and SoA_Pull_Full_Tiling) with different domain sizes. As shown, the peak performance of 1210.63 MLUPS is achieved by the SoA_Pull_Full_Tiling with domain size 256^3, where the speedup of 136 is also achieved.

(vii) Table 4 compares the performance of our work with the previous work conducted by Mawson and Revell [12]. Both implementations were conducted on the same K20 GPU. Our approach performs better than [12] from 14% to 19%. Our approach incorporates

Table 3: Performance (MLUPS) comparisons of four implementations.

Domain sizes	TimeSteps	Serial	AoS	SoA_Pull_Only	SoA_Pull_Full_Tiling
64^3	1000	9.89	111.73	759.52	1034
	5000	9.75	112.24	814.01	1115.63
	10000	9.82	111.73	818.36	1129.32
	Avg. Perf.	**9.82**	**112.41**	**797.30**	**1092.99**
128^3	1000	7.64	78.58	798.21	1115.2
	5000	7.65	78.74	855.55	1189.15
	10000	6.98	78.74	861.42	1199.33
	Avg. Perf.	**7.42**	**78.69**	**838.39**	**1167.69**
192^3	1000	9.47	76.79	811.35	1114.96
	5000	9.69	76.89	866.76	1185.95
	10000	9.56	76.91	871.39	1205.48
	Avg. Perf.	**9.57**	**76.86**	**849.83**	**1168.8**
256^3	1000	8.99	74.74	787.76	1113.77
	5000	8.91	75.09	873.8	1182.33
	10000	8.9	775.14	883.56	1210.63
	Avg. Perf.	**8.93**	**74.99**	**848.37**	**1168.91**

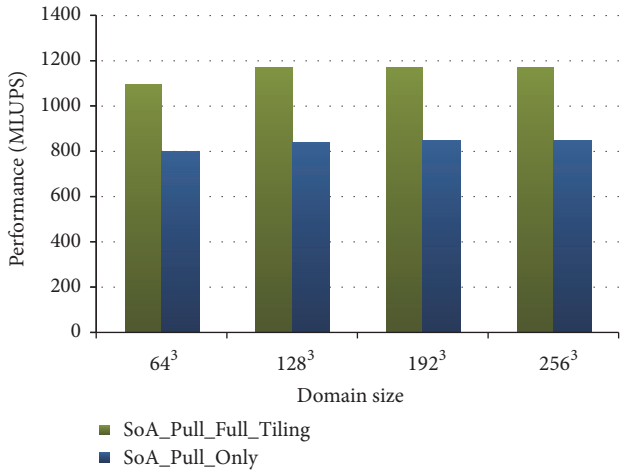


Figure 20: Performance (MLUPS) comparison of the SoA_Pull_Only and the SoA_Pull_Tiling with different domain sizes.

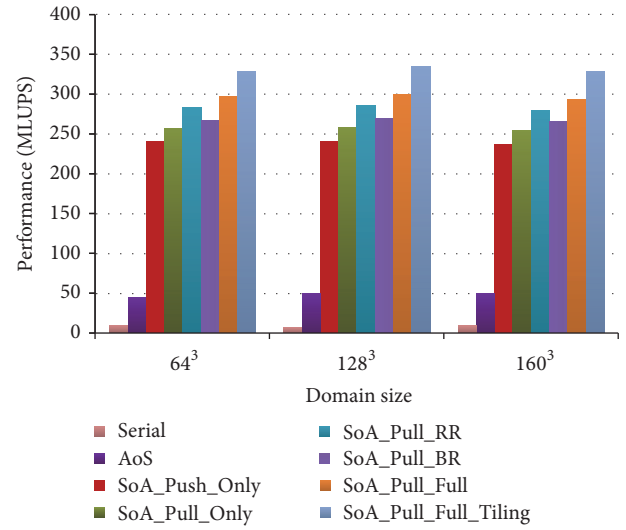


Figure 21: Performance (MLUPS) on GTX285 with different domain sizes.

Table 4: Performance (MLUPS) comparison of our work with previous work [12].

Domain sizes	Mawson and Revell (2014)	Our work
64^3	914	1129
128^3	990	1199
192^3	1036	1205
256^3	1020	1210

more optimization techniques such as the tiling optimization with the data layout change, the branch divergence removal, among others compared with [12].

(viii) In order to validate the effectiveness of our approach, we conducted more experiments on the other GPU, Nvidia GTX285. Table 5 and Figure 21 show the average performance of our implementations with domain sizes 64^3, 128^3, and 160^3. (The grid sizes larger than 160^3 cannot be accommodated in the device memory of the GTX 285.) As shown, our optimization technique, SoA_Pull_Full_Tiling, is better than the previous SoA_Pull_Only up to 22.85%. Also we obtained 46-time speedup compared with the serial implementation. The level of the performance improvement and the speedup are, however, lower on the GTX 285 compared with the K20.

Table 5: Performance (MLUPS) on GTX285.

Domain sizes	Serial	AoS	SoA_Push_Only	SoA_Pull_Only	SoA_Pull_BR	SoA_Pull_RR	SoA_Pull_Full	SoA_Pull_Full_Tiling
64^3	9.82	45.22	240.97	257.36	268.45	283.24	296.73	328.87
128^3	7.42	49.85	242.12	259.04	270.58	285.68	299.79	335.77
160^3	9.57	50.15	237.58	254.18	266.25	279.59	294.26	328.62

6. Previous Research

Previous parallelization approaches for the LBM algorithm focused on two main issues: how to efficiently organize the data and how to avoid the misalignment in the streaming (propagation) phase of the LBM. In the data organization, the AoS and the SoA are two most commonly used schemes. While AoS scheme is suitable for the CPU architecture, SoA is a better scheme for the GPU architecture when the global memory access coalition technique is incorporated. Thus, most implementations of the LBM on the GPU use the SoA as the main data organization.

In order to avoid the misalignment in the streaming phase of the LBM, there are two main approaches. The first proposed approach uses the shared memory. Tölke in [9] used the approach and implemented the D2Q9 model. Habich et al. [8] followed the same approach for the D3Q19 model. Bailey et al. in [16] also used the shared memory to achieve 100% coalescence in the propagation phase for the D3Q13 model. In the second approach, the pull scheme was used instead of the push scheme without using the shared memory.

As observed, the main aim of using the shared memory is to avoid the misaligned accesses caused by the distribution values moving to the east and west directions [8, 9, 17]. However, the shared memory implementation needs extra synchronizations and intermediate registers. This lowers the achieved bandwidth. In addition, using the shared memory limits the maximum number of threads per thread block because of the limited size of the shared memory [17] which reduces the number of active WARPs (occupancy) of the kernels, thereby hurting the performance. Using the pull scheme, instead, there is no extra synchronization cost incurred and no intermediate registers are needed. In addition, the better utilization of the registers in the pull scheme leads to generating a larger number of threads as the total number of registers is fixed. This leads to better utilization of the GPU's multithreading capability and higher performance. Latest results in [12] confirm the higher performance of the pull scheme compared with using the shared memory.

Besides the above approaches, in [12], the new feature of the Tesla K20 GPU, shuffle instruction, was applied to avoid the misalignment in the streaming phase. However, the obtained results were worse. In [18], Obrecht et al. focused on choosing careful data transfer schemes in the global memory instead of using the shared memory in order to solve the misaligned memory access problem.

There were some approaches to maximize the GPU multiprocessor occupancy by reducing the register uses per thread. Bailey et al. in [16] showed 20% improvement in maximum performance compared with the D3Q19 model in [8]. They set the number of registers used by the kernel below a certain limit using the Nvidia compiler flag. However, this approach may spill the register data to the local memory. Habich et al. [8] suggested a method to reduce the number of registers by using the base index, which forces the compiler to reuse the same register again.

A few different implementations of the LBM were attempted. Astorino et al. [19] built a GPU implementation framework for the LBM valid for the two- and three-dimensional problems. The framework is organized in a modular fashion and allows for easy modification. They used the SoA scheme and the semidirect approach as the addressing scheme. They also adopted the swapping technique to save the memory required for the LBM implementation. Rinaldi et al. [17] suggested an approach based on the single-step algorithm with a reversed collision-propagation scheme. They used the shared memory as the main computational memory instead of the global memory. In our implementation, we adopted these approaches for the SoA_Pull_Only implementation shown in Section 5.

7. Conclusion

In this paper, we developed high performance parallelization of the LBM algorithm with the D3Q19 model on the GPU. In order to improve the cache locality and minimize the overheads associated with the uncoalesced accesses in moving the data to the adjacent cells in the streaming phase of the LBM, we used the tiling optimization with the data layout change. For reducing the high register pressure for the LBM kernels and improving the available thread parallelism generated at the run time, we developed techniques for aggressively reducing the register uses for the kernels. We also developed optimization techniques for removing the branch divergence. Other already-known techniques were also adopted in our parallel implementation such as combining the streaming phase and the collision phase into one phase to reduce the memory overhead, a GPU friendly data organization scheme so-called the SoA scheme, efficient data placement of the major data structures in the GPU memory hierarchy, and adopting a data update scheme (pull scheme) to further reduce the overheads of the uncoalesced accesses. Experimental results on the 6-core 2.2 Ghz Intel Xeon processor and the Nvidia Tesla K20 GPU using CUDA show that our approach leads to impressive performance results. It delivers up to 1210.63 MLUPS throughput performance and achieves up to 136-time speedup compared with a serial implementation running on single CPU core.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (NRF-2015M3C4A7065662). This work was supported by the Human Resources Program in Energy Technology of the Korea Institute of Energy Technology Evaluation and Planning (KETEP), granted financial resource from the Ministry of Trade, Industry & Energy, Republic of Korea (no. 20154030200770).

References

[1] J.-P. Rivet and J. P. Boon, *Lattice Gas Hydrodynamics*, vol. 11, Cambridge University Press, Cambridge, UK, 2005.

[2] J. Tölke and M. Krafczyk, "TeraFLOP computing on a desktop PC with GPUs for 3D CFD," *International Journal of Computational Fluid Dynamics*, vol. 22, no. 7, pp. 443–456, 2008.

[3] G. Crimi, F. Mantovani, M. Pivanti, S. F. Schifano, and R. Tripiccione, "Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor," in *Proceedings of the 13th Annual International Conference on Computational Science (ICCS '13)*, vol. 18, pp. 551–560, Barcelona, Spain, June 2013.

[4] M. Stürmer, J. Götz, G. Richter, A. Dörfler, and U. Rüde, "Fluid flow simulation on the Cell Broadband Engine using the lattice Boltzmann method," *Computers & Mathematics with Applications*, vol. 58, no. 5, pp. 1062–1070, 2009.

[5] NVIDA, CUDA Toolkit Documentation, September 2015, http://docs.nvidia.com/cuda/index.html.

[6] GROUP KHRONOS, OpenCL, 2015, https://www.khronos.org/opencl/.

[7] OpenACC-standard.org, OpenACC, March 2012, http://www.openacc.org/.

[8] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA," *Advances in Engineering Software*, vol. 42, no. 5, pp. 266–272, 2011.

[9] J. Tölke, "Implementation of a Lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA," *Computing and Visualization in Science*, vol. 13, no. 1, pp. 29–39, 2010.

[10] G. Wellein, T. Zeiser, G. Hager, and S. Donath, "On the single processor performance of simple lattice Boltzmann kernels," *Computers and Fluids*, vol. 35, no. 8-9, pp. 910–919, 2006.

[11] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, "Comparison of different propagation steps for lattice Boltzmann methods," *Computers and Mathematics with Applications*, vol. 65, no. 6, pp. 924–935, 2013.

[12] M. J. Mawson and A. J. Revell, "Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs," *Computer Physics Communications*, vol. 185, no. 10, pp. 2566–2574, 2014.

[13] N. Tran, M. Lee, and D. H. Choi, "Memory-efficient parallelization of 3D lattice boltzmann flow solver on a GPU," in *Proceedings of the IEEE 22nd International Conference on High Performance Computing (HiPC '15)*, pp. 315–324, IEEE, Bangalore, India, December 2015.

[14] K. Iglberger, *Cache Optimizations for the Lattice Boltzmann Method in 3D*, vol. 10, Lehrstuhl für Informatik, Würzburg, Germany, 2003.

[15] J. L. Henning, "SPEC CPU2006 Benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[16] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating lattice boltzmann fluid flow simulations using graphics processors," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP '09)*, pp. 550–557, IEEE, Vienna, Austria, September 2009.

[17] P. R. Rinaldi, E. A. Dari, M. J. Vénere, and A. Clausse, "A Lattice-Boltzmann solver for 3D fluid simulation on GPU," *Simulation Modelling Practice and Theory*, vol. 25, pp. 163–171, 2012.

[18] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, "A new approach to the lattice Boltzmann method for graphics processing units," *Computers & Mathematics with Applications*, vol. 61, no. 12, pp. 3628–3638, 2011.

[19] M. Astorino, J. B. Sagredo, and A. Quarteroni, "A modular lattice boltzmann solver for GPU computing processors," *SeMA Journal*, vol. 59, no. 1, pp. 53–78, 2012.