A D3Q19 Lattice Boltzmann Solver on a GPU Using Constant-Time Circular Array Shifting

Mohammadreza Khani* and Tai-Hsien Wu†
*Department of Electrical and Computer Engineering, Western Michigan University, Kalamazoo, Michigan, USA
†Department of Chemical and Paper Engineering, Western Michigan University, Kalamazoo, Michigan, USA
*mohammadreza.khani@wmich.edu, †tai-hsien.wu@wmich.edu

***Abstract*—Lattice Boltzmann Method (LBM), specifically Bhatnagar, Gross and Krook's version (LBM-BGK), has played a substantial role in fluid dynamics computation for a long time. Although the parallel nature of LBM-BGK algorithm makes this method a very promising candidate for an implementation on a Graphics Processing Unit (GPU), the streaming process of this algorithm, with its un-coalesced memory access pattern, decreases LBM's overall performance. In this paper, we propose a new method, Structure of Arrays Constant-time Circular Array Shifting (SOA-CCAS), implemented using CUDA for a GPU to achieve higher performance. Our experimental results for a 3D shear flow application, with periodic boundary conditions, and a lid cavity application demonstrate that SOA-CCAS method has an average speedup of 167.58X compared to a sequential C LBM-BGK implementation for the two applications. Our implementation also outperforms two similar well-known LBM GPU implementations (Tölke and Habich works) using CUDA C on average by up to 2.55X, and another state-of-the-art implementation (Tran et al.) on average by up to 1.1X for the two applications.**

***Index Terms*—GPU, Parallel computing, CUDA, Fluid Dynamics, LBM-BGK**

I. INTRODUCTION AND BACKGROUND

One method to address heat transfer and fluid dynamics problems in science and engineering is to use partial differential equations approach. However, solving these equations is usually a very complex and a time-consuming task. On the other hand, numerical calculations have opened a very sophisticated path to tackle these issues with the help of new generation computers and parallel algorithms [1]. In Lattice Boltzmann method (LBM), fluid is divided into a group of particles. These particles stream (or move) along lattice links and collide with each other at the lattice sections. Because each of the collision and streaming processes are independent at each lattice, parallel algorithms are suitable for LBM [3]. LBM is a relatively new computational fluid dynamics method; it solves the Boltzmann equation, resulting in the simulation of fluid behavior. In LBM, the fluid domain is represented by a set of regularly arranged particles, and each particle has several particle distribution functions $f_\sigma(\vec{x},t)$ and discrete velocities $\vec{e}_\sigma$, where σ is dependent on the chosen lattice model. However, in all lattice models, particles move based on the value of σ, when $\sigma=0$, particles are at rest, and when $\sigma=1$ and $\sigma=2$, particles move to their nearest neighbors and their next nearest neighbors respectively. The Boltzmann equation, which discretized by the Bhatanagar-Gross-Krook (BGK) single relaxation time model [3], [22] is written as follows:

$$f_\sigma(\vec{x}+\vec{e}_\sigma\delta t, t+\delta t) = f_\sigma(\vec{x},t) - \frac{1}{\tau}[f_\sigma(\vec{x},t) - {f_\sigma}^{eq}(\vec{x},t)] \quad (1)$$

where τ is the relaxation time, and ${f_\sigma}^{eq}(\vec{x},t)$ is the equilibrium distribution function at position $\vec{x}$ and time t. Equation (1) can be decomposed into two sub-processes, collision and streaming, as follows:

$$\text{Collision} \quad \tilde{f}_\sigma(\vec{x},t) = f_\sigma(\vec{x},t) - \frac{1}{\tau}[f_\sigma(\vec{x},t) - {f_\sigma}^{eq}(\vec{x},t)] \quad (2)$$

$$\text{Streaming} \quad f_\sigma(\vec{x}+\vec{e}_\sigma\delta t, t+\delta t) = \tilde{f}_\sigma(\vec{x},t) \quad (3)$$

where $\tilde{f}$ the post-collision state of a distribution function.
The algorithm of LBM is divided into two processes: collision and streaming shown by Equations (2) and (3). Both collision and streaming are required to solve the Boltzmann equation at each time step during simulation. In addition, the collision process just depends on the information of each node and occupies most of the computing time [3]. The equilibrium distribution function can be written as (4):

$${f^{eq}}_\sigma = \omega_\sigma \rho_f [1 + 3(\vec{e}_\sigma.\vec{u}) + \frac{9}{2}(\vec{e}_\sigma.\vec{u})^2 - \frac{3}{2}(\vec{u}.\vec{u})] \quad (4)$$

where $\vec{u}$ is the macroscopic velocity of a fluid particle. The weight coefficient ω_σ and discrete velocity $\vec{e}_\sigma$ in the D3Q19 model are given by (5) and (6):

$$\omega_\sigma = \begin{cases} \frac{1}{3} & \sigma = 0 \\ \frac{1}{18} & \sigma = 1 \\ \frac{1}{36} & \sigma = 2 \end{cases} \quad (5)$$

$$\vec{e}_\sigma = \begin{cases} (0,0,0) \\ (\pm1,0,0),(0,\pm1,0),(0,0,\pm1) \\ (\pm1,\pm1,0),(\pm1,0,\pm1),(0,\pm1,\pm1) \end{cases} \quad (6)$$

The fluid kinematic viscosity υ relates to the relaxation time τ as follows:

$$\upsilon = \frac{1}{3}(\tau - \frac{1}{2})\delta t \quad (7)$$

In addition, the fluid density ρ_f and fluid momentum density $\rho_f\vec{u}$ of a particle are indicated by (8):

$$\rho_f = \sum_\sigma f_\sigma \quad , \quad \rho_f\vec{u} = \sum_\sigma f_\sigma \vec{e}_\sigma \quad (8)$$

In this work, we developed a new optimization technique, Structure of Arrays Constant-time Circular Array Shifting (SOA-CCAS), for solving LBM using a GPU. We also combined SOA-CCAS with other existing optimization techniques to improve the performance of LBM. We used a three-dimensional LBM solver for a D3Q19 model based on an implementations by Murphy [4], and Dethier et al. [5]. In addition, we implemented two different fluid applications that use LBM, with regular domains, shear flow and lid cavity. Our results show great performance compared to CPU results and higher than previous GPU implementations [6]–[8], using NVIDIA Tesla K40c GPU. Our main contributions are as follows:

- Introducing a new method to optimize LBM that parallelizes the Dethier et al's Constant time Circular Array Shifting technique (CCAS), which was developed for CPUs, for a GPU-CUDA platform to achieve higher performance for solving a 3D LBM.
- Designing a unique data structure that maps nodes on a 3D lattice into a single one-dimensional (1D) array instead of using multiple 1D arrays for all directions of the distribution functions (f^{eq}) to avoid un-coalesced memory accesses. In addition, accessing each element in the 3D lattice is explicit this way using our 1D array, which removes the burden of mapping three dimensions x, y and z to one index. These significantly improve performance over previous work.
- Avoiding *if-statements* and *computational if-statements* used by Tran et al. [6], which are mandatory on boundary positions, using our unique structure (More details in Section 7).

The rest of the paper is organized as follows: Section 2 demonstrates some of the LBM applications, Section 3 discusses related recent works on LBM using GPUs. Section 4 explains Tölke et al.'s work [7] and Habich [8] LBM implementation and shows some of our CUDA C implementations based on these studies. Section 5 explains CCAS method proposed by Murphy [4] and the Dethier et al. [5] Section 6 discusses our new optimized SOA-CCAS method. Section 7 shows our results, and finally section 8 concludes the paper.

II. LBM APPLICATIONS

The following cases are studied for LBM algorithm for a simple shear flow and a lid cavity applications:

A. Simple Shear Flow Application and Its Boundary Conditions

A simple shear flow is the case of two moving parallel plates separated by a distance h, as shown in Fig. 1. The shear rate $\dot{\gamma}$ in this example is defined in Equation (9),

$$\dot{\gamma} = \frac{2u_0}{h} \tag{9}$$

where u_0 is the desired velocity, and h is the distance between plates. To implement the shear flow algorithm, the boundary

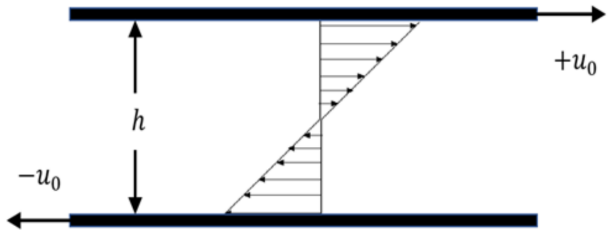

Figure 1. A simple shear flow.

conditions in the Z-direction are:

$$\begin{aligned} u_y(z=0) &= -u_0 \\ u_y(z=NZ+1) &= +u_0 \end{aligned} \tag{10}$$

where NZ is the number of grids in Z-direction, and h equals $NZ+2$. Equation (10) signifies that the fluid velocities in the Y-direction of all particles at $z=0$ and $z=NZ+1$ are set to $-u_0$ and $+u_0$, respectively.

To simulate an infinite domain in X- and Y-directions, the periodic boundary conditions described by (11) and (12) are utilized.

$$\begin{aligned} f_\sigma(x=0, t+\delta t) &= f_\sigma(x=NX, t) \\ f_\sigma(x=NX+1, t+\delta t) &= f_\sigma(x=1, t) \end{aligned} \tag{11}$$

$$\begin{aligned} f_\sigma(y=0, t+\delta t) &= f_\sigma(x=NY, t) \\ f_\sigma(y=NY+1, t+\delta t) &= f_\sigma(y=1, t) \end{aligned} \tag{12}$$

Equations (11) and (12) indicate that fluid particles leaving the fluid domain will re-enter it from the opposite side, where NX and NY above represent the numbers of grids in X- and Y-directions. A simple shear flow between two infinite large plates can be simulated by applying the above boundary conditions. For simplicity, we assumed that $NX = NY = NZ = n$ throughout this work.

B. 3D Lid Driven Cavity Flow and Its Boundary Conditions

To implement the 3D lid driven cavity flow, as shown in Fig. 2, the boundary conditions in X-, Y-, Z- direction are as follows:

$$u(x=0) = u(x=L) = (0,0,0) \tag{13}$$

$$u(y=0) = u(y=L) = (0,0,0) \tag{14}$$

$$u(z=0) = (0,0,0), \quad u(z=L) = (u_0,0,0) \tag{15}$$

where L is the edge of a cubic cavity, and u_0 is the top lid velocity. Equation (16) shows the Reynolds number (Re) in the lid-cavity flow.

$$Re = \frac{u_0 L}{\upsilon} \tag{16}$$

In this study, we set $u_0 = 0.1$.

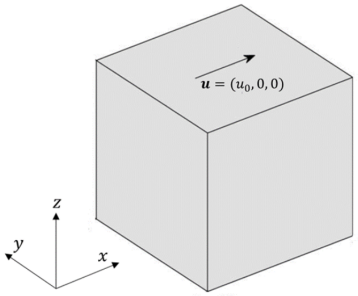

Figure 2. A sketch of 3D-lid driven cavity flow. Only the top plate has a velocity u_0 in X-direction, the rest are stationary.

III. RELATED WORK

Several studies on improving memory performance for a GPU implementation of LBM have been carried out [12]–[17], [24]–[26]. Most of these studies address data organization and memory alignment techniques. Tran et al. [6] parallelized LBM on a GPU by incorporating memory-efficient techniques such as tiling optimization and *pull* scheme, which is a data update pattern in the memory for avoiding un-coalesced accesses. In the pull scheme, three stages happen consecutively: reading distribution functions from adjacent cells, calculating ρ, u, and f^{eq} and updating values to the current cell. Furthermore, because *if-statements* cause branch divergence, they are replaced with *computational* if-statements. In [6], to avoid branch divergence also an extra virtual layer, a $ghostlayer$, was added to the boundary conditions in the streaming stage. In addition, Tran et al. used register and double floating-point reduction techniques by reusing variables as much as possible. This strategy allowed them to reach 7.1% to 8.9% of performance improvement for LBM compared to a LBM implementation that uses multiple variables. Tölke and Krafczyk [7] implemented D3Q13 model of LBM on a GPU, and they obtained up to 20X speedup compared to a PC. Tölke [9] proposed a solution to the misalignment problem (mismatching between thread number and memory access) using one-dimensional blocks and a shared memory. Tölke and Krafczyk showed in [7] that the bandwidth efficiency depends on memory access patterns. Thus, device memory access patterns should be organized into a single coalesced memory access, otherwise memory bandwidth slows down [10].

Rayoo et al. in [11] re-wrote 470.lbm code form the SPEC CPU2006 benchmark suite using CUDA C for GPUs [19]. In their work, they presented optimization techniques that include efficient code schemes, hiding latency through efficient utilization of CUDA threads, and increasing global memory bandwidth capacity using local memories for the GeForce 8800 GTX architecture using CUDA. Rayoo et al. changed an Array of Structures (AOS) scheme of the code to a Structure of Arrays (SOA), but they did not address the misalignment problem caused by the streaming stage in the LBM. Furthermore, they showed that the low number of global memory accesses, from kernels after optimization, can result in a higher speedup over sequential execution using a CPU [11].

Dethier et al. [5] improved the constant time array shifting technique introduced by Murphy [4], which decreased the total execution time for LBM algorithm and optimized memory utilization compared to a naïve implementation of LBM. Iglberger [21] presents different cache optimization techniques using 3- and 4-way blocking and discusses how correct block size can affect the performance of LBM algorithm significantly.

IV. POPULAR LBM PARALLELIZATION METHODS: TÖLKE ET AL. AND HABICH ET AL.'S WORKS

Tölke and Krafczyk [7] and Habich et al. [8] separately introduced very efficient 3D-LBM kernels (3DQ13 model) using CUDA platform developed by NVIDIA. In their work, they obtained more than 10X speedup compared to standard CPUs. The following code segments demonstrate *kernel* functions in CUDA C for density and collision operator for each node in a shear flow application with periodic boundary conditions, which are developed based on Tölke and Krafczyk [7] and Habich et al. [8] works using our data structure. In their implementation, a single fluid node is assigned to each CUDA thread and all threads of a block will handle fluid nodes of the same y and z stripe [8].

1) Fluid density ρ for each node with periodic boundary conditions:

```
__global__ void rho_sum(double *f, double *rho){

int tid = threadIdx.x + n*blockIdx.x + n*n*blockIdx.y;
double rho_sum = 0;
// tid is the position of each node in 1D array f
// i is the fluid velocity in direction ith
// threadIdx.x is assigned to each node
// fluid density computation on the boundary
// of fluid is avoided by if-statment
if (blockIdx.x != 0 && blockIdx.x != (n - 1)
        && threadIdx.x != 0 && threadIdx.x != (n - 1)
        && blockIdx.y !=0 && blockIdx.y !=(n-1)){
        for (int i = 0; i < 19; i++)
                rho_sum += f[N * i + tid];
                rho[tid] = rho_sum;
        }
}
```

2) Collison operator for each node with periodic boundary conditions:

```
__global__ void f_postcolli(double *f, double *feq){
const double tau = 1; // relaxation time
int tid = threadIdx.x + n*blockIdx.x + n*n*blockIdx.y;
// tid is the position of each node in 1D array f
// i is the fluid velocity in direction ith
// threadIdx.x is assigned to each node
// collision computation on the boundary of fluid
//   is avoided by if-statment
if (blockIdx.x != 0 && blockIdx.x != (n - 1)
         && threadIdx.x != 0 && threadIdx.x != (n - 1)
         && blockIdx.y != 0 && blockIdx.y != (n - 1)){
        for (int i = 0; i < 19; i++)
        f[N * i + tid] = (double)f[N * i + tid]
        - (1 / tau)*(f[N *i + tid] - feq[N * i + tid]);
}
}
```

V. SOME CPU-BASED SEQUENTIAL LBM IMPLEMENTATIONS

The following data organization and data access optimization techniques are proposed by Murphy [4] and Dethier et al. [5] based on data locality principles for a single-thread CPU implementation.

1) **Murphy's Method**
In Murphy's method, a multidimensional array is translated to a one-dimensional (1D) array as shown in Fig. 3 for a 3×3 array example.

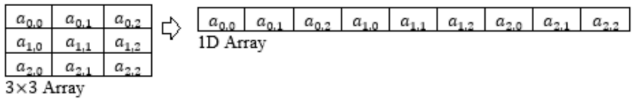

Figure 3. Array translation.

This method utilizes a constant-time shift of an array's elements. The shift procedure is performed by shifting the base pointer of the array to the left and right. Shifting k positions to right or left causes k elements of the array at the beginning or at the end to become undefined. Murphy [4] defines a shift of elements of an array with an integer variable offset as the following:

$$\begin{cases} offset > 0 & \text{shifted right from position } i \text{ to } i+k \\ offset < 0 & \text{shifted left from position } i \text{ to } i+k \\ offset = 0 & \text{no shift} \end{cases} \quad (17)$$

where i is the position of each element. In the case of shifting elements of a multidimensional array, an offset vector is assigned [5]. Fig. 4 shows the shift operation for a 3×3 array with an offset vector of (1,1). 1D array representation utilizes a translation-offset vector. Fig. 5 illustrates the shift of 1D array for the 3×3 array example.

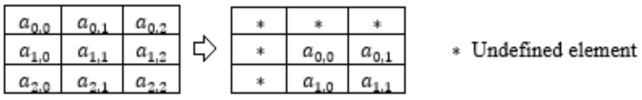

Figure 4. 3×3 array shifted using an offset vector (1, 1)

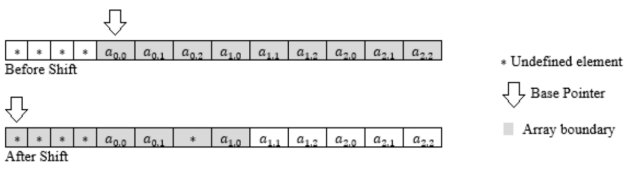

Figure 5. Shifting of a vector, representation of 3×3 array.

In Murphy's method, to shift elements to the left and to the right without overwriting other elements of the array, enough memory allocation that depends on the size of the lattice is required on both sides of the array. After reaching the maximum allocated memory around the array, copying elements of the array to their initial positions and resetting the base pointer is unavoidable (memory overhead and pointer reset issues).

2) **Dethier et al.'s Method**
Dethier et al. [5] introduced a method that eliminates memory overhead and pointer reset issues with Murphy's method, but it adds some computational complexity to the problem. Their method was targeted for CPUs only, where they used a *circular array* to overcome Murphy's limitations of memory overhead and data reset operation issues. In their implementation, the base pointer never moves, but an offset pointing to the actual first position of the circular array can be moved [5]. Each element in the circular array at location i is mapped into f and f^{eq} at position $(offset + i)\ mod\ M$. Equation (18) is executed to apply an offset K to the elements of f and f^{eq}:

$$offset = (offset - K) \quad mod \quad M \quad (18)$$

where M is the size of the circular array. Fig. 6 demonstrates the circular array shifting with one unit shift (red line symbolizes the pointer).

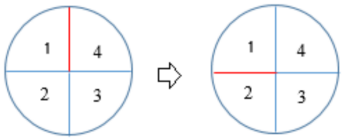

Figure 6. The circular array shifting with one unit shift.

VI. PROPOSED PARALLEL GPU-BASED CONSTANT-TIME CIRCULAR ARRAY SHIFTING (CCAS) METHOD

Although Dethier et al. [5] introduced an efficient way to implement the streaming stage that eliminates overhead and pointer reset, which were required in Murphy's method, data locality problems and using an additional modulus operator to implement CCAS, increases the computational time of the collision stage. To tackle these issues, Dethier et al. [5] utilized lattice-block copy and block-lattice copy schemes which decrease the number of cache misses as well as the computational time of the collision stage for a CPU. In Dethier et al's technique, accessing λ subsequent particle distribution functions (pdfs), $f[ni+(t+offset[i]\ mod\ s]$ with $i = 0, \ldots, 18$ (s is the size of 1D representation of 3D array containing the pdfs and $t = I, \ldots, I+\lambda$ where I is a positive integer number), provides continuous memory accesses for better data locality and less cache misses. In our work, we parallelized CCAS on a GPU-CUDA platform and combined it to the optimizations done by Tran et al. [6]. Below is the description of our implementation.

1) We used a SOA data organization to store the values of the pdfs in a 1D array, so that different threads can access consecutive memory locations.

2) To avoid memory latency in transferring the pdfs and saving memory space in global memory of the GPU in the streaming stage, we used two arrays based on Dethier et al's method; *offset* and *dev_offset*. These arrays hold the required offset for each direction of pdfs. At the beginning of the program, the values in the offset array are initialized to zeros. Then after the execution of the streaming step during the first iteration, each pdf is assigned a specific offset, based on the type of its lattice's direction as shown in Table 1. These values are copied into *dev_offset* array of the GPU global memory.

Table I
OFFSET FOR EACH PDF DIRECTION

Direction of each lattice pdfs	Offset value	Direction of each lattice pdfs	Offset value
0	0	10	$n+1$
1	-1	11	$-(n.n+1)$
2	+1	12	$-(n.n-1)$
3	$-n$	13	$n.n-1$
4	$+n$	14	$n.n+1$
5	$-n.n$	15	$-n.(n+1)$
6	$+n.n$	16	$-n.(n-1)$
7	$-(n+1)$	17	$n.(n-1)$
8	$-(n-1)$	18	$n.(n+1)$
9	$n-1$		

3) The streaming stage of the LBM-BGK (D3Q19) is eliminated and is combined into *rho_sum*, *u_sum*, and *f_equilibrium* kernels in the form of an offset. This modification can avoid the updating f and f^{eq} time and improve the performance impressively. In addition, in each iteration, the information of the adjacent nodes is read based on the offset modification. After ρ, u, and f^{eq} are calculated. Then in collision stage the pdfs are updated. Since writing to memory, in our implementation, happens only in *f_post_collision* kernel and boundary kernels, this results in much lower latency than other methods. The following code shows a *rho_sum* kernel based on an offset vector modification in CUDA C:

```
rho_sum += f[N*i+(tid+offset[i])]
```

4) To maximize the performance of SOA-CCAS, we also implemented similar techniques as proposed by Tran et al. [6]:
   a) As discussed in the introduction, LBM's algorithm requires two stages (kernels): *f_post_collision* and *streaming*. SOA-CCAS helps the overall performance because it avoids extra loads/stores from/to the global memory and CUDA synchronization cost by combining the two stages and exploiting the offset vector.
   b) Similar techniques were implemented like those in Tran et al. such as decreasing the number of used registers, 3D tiling data layout and finally pull scheme (reading pdfs from adjacent cells) to achieve high performance in SOA-CCAS.

5) Thread divergence, which happens because of *if-statements* in LBM algorithm, can be one of the major obstacles in decreasing performance of a CUDA-GPU platform [8]. Tran et al. [6] used the *computational-if-statement* to replace control instructions by computational equivalent statements. On the other hand, in our implementation, streaming at the boundary positions for the SOA-CCAS does not require *computational-if-statement* by purposely allowing streaming to happen on boundary positions and inside the fluid. Although streaming should only happen inside the fluid for correct results normally, we fix this issue by applying boundary kernels to these locations. This approach increases the performance and avoids both if-statements, and *computational-if-statement* to eliminate thread divergence.

VII. VALIDATION OF THE PROPOSED METHOD

We already validated our parallelized implementation of LBM-BGK by comparing the results of our SOA-CCAS method with the results of our previous work [20] for the shear flow application with periodic boundary conditions. In this paper, we show our validation results for SOA-CCAS for the lid-cavity LBM application only. To validate the lid-cavity application, we compared our results with the results of Santanu De et al. [18]. Fig. 7 demonstrates the normalized steady-state velocity profiles computed from the proposed SOA-CCAS method. The figure shows that for $N = 36 \times 36 \times 36$, Reynold number $Re = 100$, and lid-velocity $u = 0.1$, the normalized x- and y- components of velocity profiles along the center-planes, $(\frac{N_x}{2}, y, \frac{N_z}{2})$ and $(x, \frac{N_y}{2}, \frac{N_z}{2})$, agree well with results of Santanu De et al. [18] at the similar conditions for a 3D lid-driven cavity.

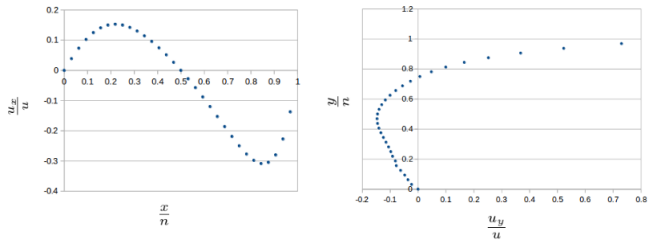

Figure 7. Normalized steady-state velocity profiles for; (a) $u_x(x, \frac{1}{2}, y, \frac{1}{2})$.

VIII. RESULTS AND DISCUSSION

In this work, we used a Nvidia Tesla K40c GPU with 12GB of GDDR5 memory with 288GHz bandwidth, 2880 processor cores - each core has 745MHz base clock, and PCI Express Gen3 $\times 16$ connected to a 6-core CPU (Intel Xeon E5-2650 2.0GHz). The system we used runs Linux CentOS 6. We implemented both 3D shear flow and lid cavity applications as test cases for our accelerated LBM-BGK algorithm with the D3Q19 model [7] using CUDA C. The number of the

time steps used in the experiments is set to 1000. We also set the domain lattice sizes to 64^3, 128^3, and 192^3. To compare the performance of the various techniques we used MLUPS (Million Lattice Updates Per Second) computed similar to [16] and [22]:

$$\frac{NX \times NY \times NZ \times \text{Number of loops} \times 10^{-6}}{\text{Total time}}. \quad (19)$$

Fig. 8 and 9 compare the performance of our sequential C implementation (using a single CPU core) and our proposed SOA-CCAS GPU implementation for 64^3 , 128^3, and 192^3 domain sizes for the 3D shear flow and lid cavity applications respectively.

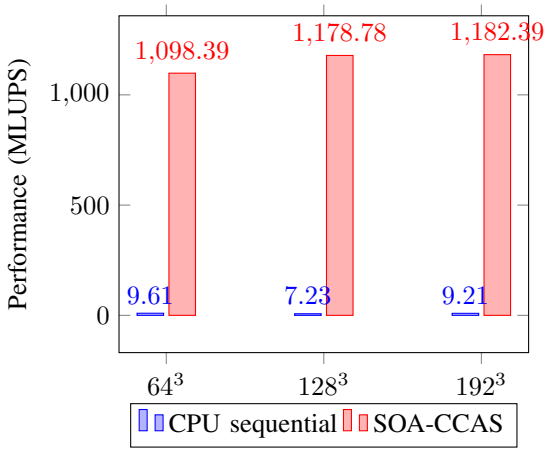

Figure 8. Performance in MLUP, SOA-CCAS compared to CPU sequential results for three domain sizes for 3D shear flow.

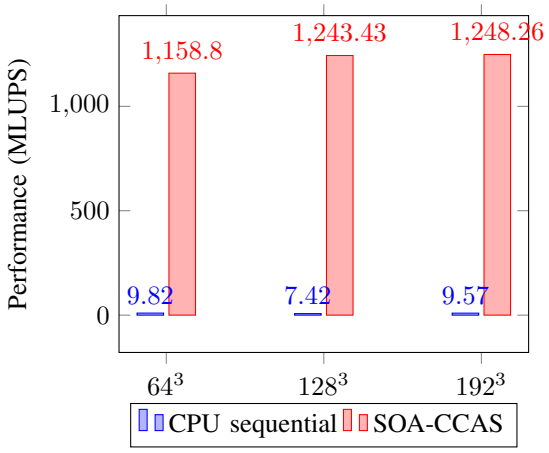

Figure 9. Performance in MLUP, SOA-CCAS compared to CPU sequential results for three domain sizes for 3D lid cavity.

SOA-CCAS outperforms one CPU's sequential code from 114.29X to 163X for shear flow and from 118X to 167.58X for lid cavity, depending on the domain size. We also implemented and compared the Tölke et al. [7] and Tran et al's [6] optimizations combined GPU implementation and compared with our SOA-CCAS. The results for this comparison are shown in Fig. 10 for the shear flow and Fig. 11 for the lid cavity applications. SOA-CCAS has on average 14.39% MLUPS improvement over Tölke et al. for the shear flow and lid cavity applications, depending on the domain size. SOA-CCAS is faster than Tran et al. method on average by up to 7.94% for both cases. Because in shear flow the computations of the boundary conditions are more complex, we were able to benefit more from our optimizations in the lid cavity case and achieve a higher speedup.

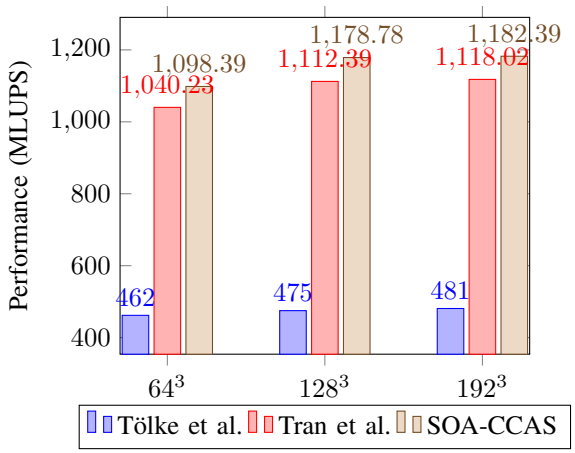

Figure 10. Performance comparison in MLUPS for SOA-CCAS with Tölke et al. [7], and Tran et al. [6] for 3D Shear flow.

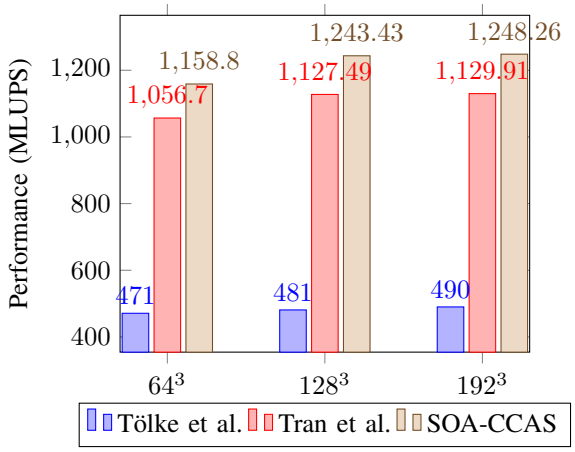

Figure 11. Performance comparison in MLUPS for SOA-CCAS with Tölke et al. [7], and Tran et al. [6] for 3D lid cavity.

IX. CONCLUSION

In this work, we introduced a new technique, SOA-CCAS (Structure of Arrays Constant-Time Circular Array Shifting), to LBM-BGK D3Q19 model using CUDA C on a GPU platform to achieve higher performance compared to other state of the art techniques. We compared our results to previous well-known and new techniques that optimize LBM using CUDA. Our results on a Nvidia Tesla K40c GPU accelerator demonstrate up to 167.58X, 2.55X, and 1.1X improvement in

performance compared to single CPU, Tölke et al. [7] and Tran et al. [6] implementations of LBM respectively. While our work focuses on fluid applications only, we plan to modify our method in future to work with irregular and porous domains.

X. ACKNOWLEDGMENT

We would also like to show our gratitude to NVIDIA corporation for donating a NVIDIA Tesla K40 GPU accelerator to conduct this research.

REFERENCES

[1] A. A. Mohamad, "Lattice Boltzmann Method, Fundamentals and Engineering Applications with Computer Codes," Springer London, (2011) J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[2] Y. H. Qian, D. d'Humières, and P. Lallemand, "Lattice BGK Models for Navier-Stokes Equation," EPL (Europhysics Letters), 17(6), 479 (1992).

[3] P. L. Bhatnagar, E. P. Gross, and M. Krook, "A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-component Systems, ". Physical review, 94(3), 511 (1954).

[4] S. Murphy, "Performance of Lattice Boltzmann Kernels," Master's thesis, University of Edinburgh, this document is available at http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0405/0762240-9jdissertation1.1.pdf (2005).

[5] G. Dethier, P. A. Marneffe, and P. Marchot, "Lattice Boltzmann Simulation Code Optimization Based on Constant-time Circular Array Shifting", International Conference on Computational Science, ICCS 2011.

[6] N. Tran, M. Lee, and D. Choi, "Memory-Efficient Parallelization of 3D Lattice Boltzmann Flow Solver on a GPU," IEEE 22nd International Conference on High Performance Computing, 2015.

[7] J. Tölke, and M. Krafczyk, "TeraFLOP Computing on A Desktop PC with GPUs for 3D CFD," International Journal of Computational Fluid Dynamics, 22(7), 443-456. 2008

[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, 42(5), 266-272 (2011).

[9] J. Tölke, "Implementation of a Lattice Boltzmann Kernel Using the Compute Unified Device Architecture Developed by NVidia," Computer Visual Science, 2010, 13:29-39.

[10] Тутубалина Алексея, "8800 GTX Performance Tests," http://blog.lexa.ru/2007/03/08/nvidia_8800gtx_skorost__chtenija_tekstur.html.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp.73-82.

[12] M. Mawson, A. Revell, "Memory Transfer Optimization for Lattice Boltzmann Solver on Kepler Architecture NVidia GPUs," Computer Physics Communications 185 (2014) 25662574.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J. Roux , "A New Approach to the Lattice Boltzmann Method for Graphics Processing Units," Computers and Mathematics with Applications 61 (2001) 3628-3638.

[14] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors," 2008.

[15] M. Boyer, "CUDA Optimization Techniques," LAVA Lab CUDA Support, University of Virginia.

[16] Xian W., Takayuki A., "Multi-GPU Performance of Incomperssible Flow Computation by Lattice Boltzmann Method on GPU Cluster," Parallel Computing Vol. 37 (2011) 521-535.

[17] M. Astorino, J. Becerra Sagredo, and A. Quarteroni, "A Modular Lattice Boltzmann Solver for GPU Computing Processors," MATHICSE Technical Report, EPFL - SB – MATHICSE, 2011.

[18] S. De, K. Nagendra, K. Lakshmisha, "Simulation of Laminar Flow in a Three-dimensional Lid-driven Cavity by Lattice Boltzmann Method," International Journal of Numerical Methods for Heat & Fluid Flow, 790, May 2008

[19] https://www.spec.org/cpu2006/

[20] Tai-Hsien Wu, Mohammadreza Khani, Lina Sawalha, James Springstead, John Kapenga, and Dewei Qi, "A CUDA-Based Implementation of a Fluid-Solid Interaction Solver: The Immersed Boundary Lattice-Boltzmann Lattice-Spring Method," Communications in Computational Physics (CiCP), Cambridge University Press 2017.

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

[22] Nhat-Phuong Tran, Myungho Lee, and Sugwon Hong, "Performance Optimization of 3D Lattice Boltzmann Flow Solver on a GPU," Scientific Programming Volume 2017, Article ID 1205892.

[23] Renwei Mei, Wei Shyy, Dazhi Yu, Li-Shi Luo, "Lattice Boltzmann Method for 3-D Flows with Curved Boundary," NASA/CR-2002-211657. NASA Center for AeroSpace Information (CASI), 7121 Standard Drive, Hanover, MD 21076-1320, June 2002.

[24] N. Delbosc, J. L. Summers, A. I. Khan, N. Kapur, C. J. Noakes, "Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation," Computers and Mathematics with Applications 67 (2014) 462–475.

[25] Qinlong Ren, Cho Lik Chan, "Numerical study of double-diffusive convection in a vertical cavity with Soret and Dufour effects by lattice Boltzmann method on GPU," International Journal of Heat and Mass Transfer Volume 93, February 2016, Pages 538-553.

[26] Ye Zhao, "Lattice Boltzmann based PDE solver on the GPU," The Visual Computer, International Journal of Computer Graphics, (2008) 24: 323–333.