Received: 16 January 2017 | Revised: 3 April 2017 | Accepted: 25 May 2017

DOI: 10.1002/cpe.4221

WILEY

SPECIAL ISSUE PAPER

Reducing memory requirements for large size LBM simulations on GPUs

Pedro Valero-Lara

Barcelona Supercomputing Center (BSC), Barcelona, Spain

Correspondence
Pedro Valero-Lara, Barcelona Supercomputing Center (BSC), Barcelona, Spain.
Email: pedro.valero@bsc.es

Funding information
Spanish Ministry of Economy and Competitiveness (MINECO): BCAM Severo Ochoa accreditation, Grant/Award Number: SEV-2013-0323, MTM2013-40824; Computación de Altas Prestaciones-VII, Grant/Award Number: TIN2015-65316-P; Basque Excellence Research Center (BERC 2014-2017)

Summary

The scientific community in its never-ending road of larger and more efficient computational resources is in need of more efficient implementations that can adapt efficiently on the current parallel platforms. Graphics processing units are an appropriate platform that cover some of these demands. This architecture presents a high performance with a reduced cost and an efficient power consumption. However, the memory capacity in these devices is reduced and so expensive memory transfers are necessary to deal with big problems. Today, the lattice-Boltzmann method (LBM) has positioned as an efficient approach for Computational Fluid Dynamics simulations. Despite this method is particularly amenable to be efficiently parallelized, it is in need of a considerable memory capacity, which is the consequence of a dramatic fall in performance when dealing with large simulations. In this work, we propose some initiatives to minimize such demand of memory, which allows us to execute bigger simulations on the same platform without additional memory transfers, keeping a high performance. In particular, we present 2 new implementations, LBM-Ghost and LBM-Swap, which are deeply analyzed, presenting the pros and cons of each of them.

KEYWORDS

Computational Fluid Dynamics, CUDA, GPU, lattice-Boltzmann method

1 | INTRODUCTION

Graphics processing units (GPUs) are today an efficient alternative to other architectures, in particular, because of their computational capacity and efficient power consumption. Many software packages have already been ported to take advantage of GPU computing, although there are some applications or solvers that can be difficult to tune[1,2] for GPUs. Fortunately, other solvers are particularly well suited for GPU acceleration and are able to achieve significant performance improvements. Lattice-Boltzmann method (LBM)[3] is one of these examples due to its inherently data-parallel nature. The parallelism is abundant in LBM, which is also amenable to fine grain parallelization. This is particularly interesting for GPU computing. The benefit of using LBM on parallel computers is consistently confirmed in many works,[4,5] for a large number of different problems and computing platforms. In one study,[6] T. Pohl et al improved the temporal locality for cache-based multicore architectures. P. R. Rinaldi et al[5] reduced the number of accesses to global memory by using a different ordering for the LBM steps causing a high impact on performance for GPU computing. The LBM has been tested in multiple parallel platforms, such as multicore,[6] hardware accelerators,[4,5,7] and distributed memory computers.[8-10] Also, we can find many tools[8,9,11,12] based on LBM, which have consolidated LBM in academia and industry. In this work, we use one of them, the LBM-HPC package.[9]

The demand of computational resources from scientific community is constantly increasing in order to simulate more and more complex scenarios. One of the most important challenges to deal with such scenarios is the large memory capacity that the scientific applications need. Multiple works have explored new techniques to reduce the impact of some applications on memory capacity.[13-17] Although LBM is amenable to be efficiently parallelized, it is in need of a high memory capacity. Our main motivation is the developing of 2 new approaches, *LBM-Ghost* and *LBM-Swap*, which minimize the demand of memory for LBM simulations over NVIDIA GPUs. In *LBM-Ghost*, we propose the use of *ghost cells* to minimize the memory requirements. The implementation of this idea is in need of nontrivial optimizations in comparison to the state-of-the-art implementations, which make difficult its implementation. The present work extends the previously published works[18,19] with additional contributions. Ihis work includes a new approach to minimize the memory demand for LBM simulations, *LBM-Swap*. This approach is based on the work developed by J. Latt,[20]

 Copyright © 2017 John Wiley & Sons, Ltd.

which is adapted to NVIDIA GPUs in the present work. Unlike the *LBM-Ghost*, the *LBM-Swap* is much easier to implement. Both initiatives allow us to execute bigger problems over the same platform, avoiding computationally expensive memory transfers.

The remainder of this paper is organized as follows. In Section 2, we introduce the general numerical and implementation framework for LBM. After that, Sections 3 and 4 describe the different optimizations and parallel strategies envisaged to achieve high performance when dealing with large simulation domains. Finally, we discuss the performance results of the proposed techniques in Section 5. We conclude in Section 6 with a summary of the main contributions of this work.

2 | LATTICE-BOLTZMANN METHOD

Most of the current solvers simulate the transport equations (heat, mass, and momentum) at macroscopic scale.[21] Otherwise, the medium can be also seen from a microscopic viewpoint where tiny particles (molecules and atoms) collide with each other (molecular dynamic).[22] In this scale, where there is no definition of viscosity, heat capacity, temperature, pressure, etc, the interparticle forces as well as the location, velocity, and trajectory of each of the particles must be computed, being extremely expensive computationally.[22] Other methods use statistical mechanisms to connect the microscopic and macroscopic worlds. The use of these methods does not require the management of every individual particle, obtaining the important macroscopic effects with manageable computer resources. This is the main idea of the Boltzmann equation and the mesoscopic scale.[22]

Previous works have compared the numerical accuracy of the LBM with respect to other methods based on Navier-Stokes (see previous studies[23,24]). They showed that LBM presents an equivalent precision over a large number of applications. There are multiple applications where LBM has been used: high Reynolds turbulent flows,[25] aeroacoustics problems,[26] and bioengineering applications,[4] among others. Also, LBM can be integrated with other methods, such as the immersed boundary method for fluid-solid interaction problems.[7,27]

The LBM combines some features of the Boltzmann equation over a finite number of microscopic speeds. The LBM presents lattice-symmetry characteristics that allow to respect the conservation of the macroscopic moments.[28] The standard LBM[29] is an explicit solver for incompressible flows. It divides each temporal iteration into 2 steps, one for propagation-advection (streaming) and one for collision (interparticle interactions), achieving a first order in time and second order in space scheme.

The LBM describes the fluid behavior at mesoscopic scale. At this level, the fluid is modeled by a distribution function of the microscopic particle (f). The LBM solves the particles speed distribution by discretizing the speed space over a discrete finite number of possible speeds. The distribution function evolves according to the following equation:

$$\frac{\partial f}{\partial t} + e\nabla f = \Omega, \tag{1}$$

where f is the particle distribution function, e is the discrete space of speeds, and Ω is the collision operator. By discretizing the distribution function f in space, in time, and in speed ($e = e_i$), we obtain $f_i(x, t)$, which describes the probability of finding a particle located at x at time t with speed e_i. $e\nabla f$ can be discretized as follows:

$$e\nabla f = e_i \nabla f_i = \frac{f_i(x + e_i\Delta t, t + \Delta t) - f_i(x, t + \Delta t)}{\Delta t}. \tag{2}$$

In this way, the particles can move only along the links of a regular *lattice* (Figure 1) defined by the next discrete speeds ($e_0 = c(0, 0)$;$e_i = c(\pm 1, 0)$, $c(0, \pm 1)$, $i = 1, \dots, 4$; $e_i = c(\pm 1, \pm 1)$, $i = 5, \dots, 8$ with $c = \Delta x/\Delta t$) so that the synchronous particle displacements $\Delta x_i = e_i \Delta t$ never takes the fluid particles away from the *lattice*. In this study, we use the standard 2-dimensional 9-speed *latticeD2Q9*.[28]

The operator Ω computes the changes caused by the collision between particles at microscopic scale, which is defined by the function (f). To calculate this operator, we consider the Bhatnagar-Gross-Krook formulation, [30] which relies upon a unique relaxation time, τ, toward the equilibrium distribution f_i^{eq}:

$$\Omega = -\frac{1}{\tau}\left(f_i(x, t) - f_i^{eq}(x, t)\right). \tag{3}$$

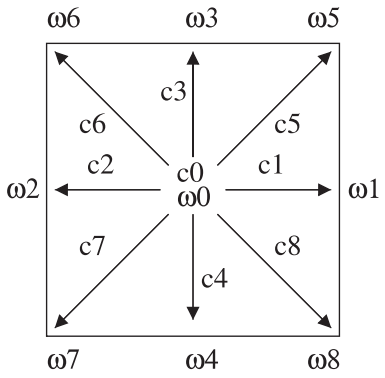

FIGURE 1 The standard 2-dimensional 9-speed lattice (*D2Q9*)[30]

The equilibrium function $f^{eq}(x,t)$ can be obtained by *Taylor* series expansion of the *Maxwell-Boltzmann* equilibrium distribution [29]:

$$f_i^{eq} = \rho\omega_i \left[1 + \frac{e_i \cdot u}{c_s^2} + \frac{(e_i \cdot u)^2}{2c_s^4} - \frac{u^2}{2c_s^2}\right], \tag{4}$$

where c_s is the speed of sound ($c_s = 1/\sqrt{3}$), u is the vertical or horizontal component (see Algorithm 1) of the macroscopic velocity in the given position, and the weight coefficients ω_i are $\omega_0 = 4/9, \omega_i = 1/9, i = 1, \ldots, 4$ and $\omega_i = 1/36, i = 5, \ldots, 8$ based on the current normalization. Through the use of the collision operator and substituting the term $\frac{\partial f_i}{\partial t}$ with a first-order temporal discretization, the discrete Boltzmann equation can be written as follows:

$$\begin{aligned} \frac{f_i(x, t+\Delta t) - f_i(x,t)}{\Delta t} + \frac{f_i(x + e_i\Delta t, t+\Delta t) - f_i(x, t+\Delta t)}{\Delta t} \\ = -\frac{1}{\tau}\left(f_i(x,t) - f_i^{eq}(x,t)\right), \end{aligned} \tag{5}$$

which can be compactly written as follows:

$$f_i(x + e_i\Delta t, t+\Delta t) - f_i(x,t) = -\frac{\Delta t}{\tau}\left(f(x,t) - f_i^{eq}(x,t)\right). \tag{6}$$

The macroscopic velocity u in Equation 4 must satisfy a Mach number requirement $|u|/c_s \approx M \ll 1$, which can be seen as the Courant Friedrichs Lewy number for classical Navier-Stokes solvers.

As mentioned above, Equation 6 is typically advanced in time in 2 stages, the collision and the streaming stages.

Given $f_i(x,t)$ compute

$$\rho = \sum f_i(x,t) \quad \text{and}$$
$$\rho u = \sum e_i f_i(x,t)$$

Collision stage:

$$f_i^*(x, t+\Delta t) = f_i(x,t) - \frac{\Delta t}{\tau}\left(f(x,t) - f_i^{eq}(x,t)\right)$$

Streaming stage:

$$f_i(x + e_i\Delta t, t+\Delta t) = f_i^*(x, t+\Delta t).$$

Algorithm 1 LBM *pull*.

1: **for** $ind = 1 \rightarrow Nx \cdot Ny$ **do**
2: **Streaming**
3: **for** $i = 0 \rightarrow 8$ **do**
4: $x_{stream} = x - c_x[i]$
5: $y_{stream} = y - c_y[i]$
6: $ind_{stream} = y_{stream} \cdot Nx + x_{stream}$
7: $f[i] = f_1[i][ind_{stream}]$
8: **end for**
9: **for** $i = 0 \rightarrow 8$ **do**
10: $\rho += f[i]$
11: $u_x += c_x[i] \cdot f[i]$
12: $u_y += c_y[i] \cdot f[i]$
13: **end for**
14: $u_x = u_x/\rho$
15: $u_y = u_y/\rho$
16: *Synchronization point (only) for our approach based on Ghost Cell*
17: *syncthreads*()
18: **Collision**
19: **for** $i = 0 \rightarrow 8$ **do**
20: $cu = c_x[i] \cdot u_x + c_y[i] \cdot u_y$
21: $f_{eq} = \omega[i] \cdot \rho \cdot (1 + 3 \cdot cu + cu^2 - 1.5 \cdot (u_x)^2 + u_y)^2$
22: $f_2[i][ind] = f[i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
23: **end for**
24: **end for**

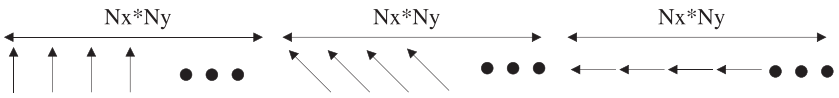

FIGURE 2 Structure of array data layout to store the discrete distribution function f_i in memory

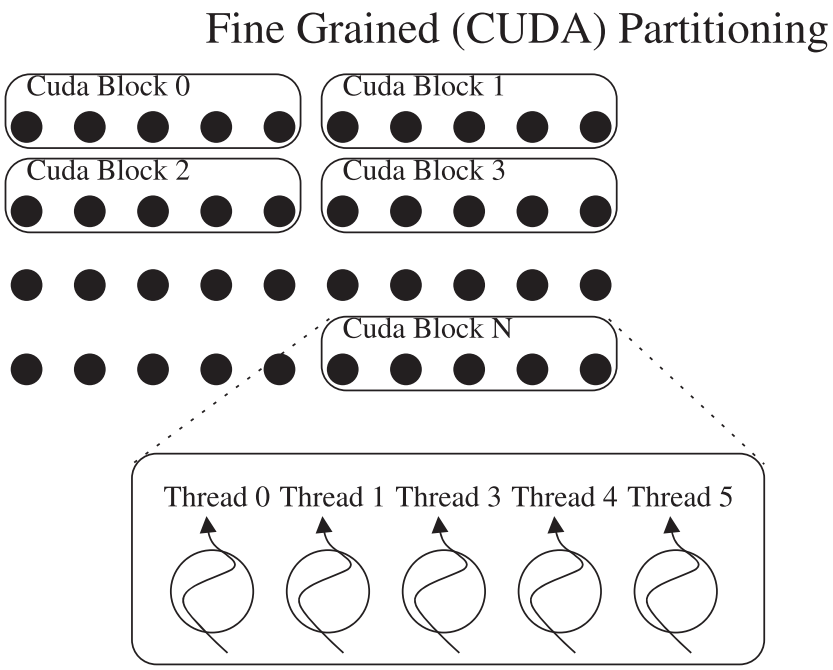

FIGURE 3 Fine-grained distributions of the *lattice* nodes

The LBM exhibits a high degree of parallelism, which is amenable to fine granularity (one thread per lattice node), as the computing of each of the *lattice* points is completely independent. To carry out LBM streaming in parallel, we need 2 different distribution functions (f_1 and f_2 in Algorithm 1).

We decided to work with the *pull* approach (introduced by Wellein et al[31]). This approach has been widely analyzed in many studies.[5,7,27] This is implemented via a single loop where each *lattice* node can be independently computed by performing one complete time step of LBM. This implementation is given in Algorithm 1. Basically, the *pull* approach fuses in a single loop (which iterates over the entire domain), the computation of both LBM operations, LBM collision, and LBM streaming, to improve temporal locality. Furthermore, it is not in need of any synchronization among these operations. Also, it minimizes the pressure on memory with respect to other approaches, as the macroscopic level can be completely computed on the highest levels of memory hierarchy (registers/L1 cache).

Memory management plays a crucial role in LBM implementations. The information of the fluid domain should be stored in memory in such way that reduces the number of memory accesses and keeps the implementation highly efficient by taking advantages of vector units. We exploit coalescence by using a structure of array approach. This idea (*pull-coalescing*) has proven to be a fast implementation in multicore and GPUs architectures.[5,7,27,32] The discrete distribution function f_i is stored sequentially in the same array (see Figure 2, where *Nx* and *Ny* are the number of horizontal and vertical fluid nodes, respectively). In this way, consecutive threads access to contiguous memory locations.

Parallelism is abundant in the LBM update and can be exploited in different ways. The recommendable parallelization of LBM over GPUs consists of using a single *kernel* by using a 1D Grid of 1D CUDA block, in which each CUDA thread performs a complete LBM update on a single *lattice* node[7]. *Lattice* nodes are distributed across GPU cores using a fine-grained distribution (Figure 3).

In order to exploit the parallelism found in the LBM, previous studies make use of 2 different data set.[5,8,9,12,27] Basically, it consists of using an *AB* scheme,[7] which holds the data of 2 successive time steps (A and B) and the simulation alternates between reading from A and writing to B, and vice versa. In this work, we propose 2 alternatives that follow an *AA* scheme to reduce such high memory requirements: one by adapting the use of *ghost cell* to LBM and one by adapting the *LBM-Swap* approach to our platform (NVIDIA GPUs).

3 | LBM-GHOST CELLS

This section explains how we have adapted the use of *ghost cells* to LBM to reduce the memory requirements for GPU-based implementations.

Although, the *ghost cells* are usually used for communication in distributed memory systems,[33] we use this strategy to reduce memory requirements and avoid race conditions among the set of CUDA blocks (*fluid blocks*). To minimize the number of *ghost cells*, we use the biggest size of CUDA

block possible. The use of *ghost cells* consists of replicating the borders of all immediate neighbors blocks, in our case *fluid blocks*. These *ghost cells* are not updated locally, but provide stencil values when updating the borders of local blocks. Every *ghost cell* is a duplicate of a piece of memory located in neighbors nodes. To clarify, Figure 4 illustrates a simple scheme for our interpretation of the *ghost cell* strategy applied to LBM (*LBM-Ghost*).

In LBM-streaming operation (Figure 5), some of the lattice speed in each *ghost cell* is used by adjacent fluid (*lattice*) elements located in neighbors *fluid blocks*. Depending on the position of the fluid units, a different pattern needs to be computed for the LBM-streaming operation. For instance, if one fluid element is located in one of the corners of the *fluid block*, this requires to take 5 lattice speed from 3 different *ghost cells*. However, if it is in other position of the boundary, it have to take 3 lattice speed from one *ghost cell* (Figure 5).

The information of the *ghost cells* has to be updated once per time step. The updating is computing via a second kernel before computing LBM. This kernel moves some lattice speed from lattice units to *ghost cells*. This CUDA kernel is computed by as many threads as *ghost cells*. To optimize memory management and minimize divergence, continuous CUDA blocks compute each of the updating cases. To clarify Figure 6 shows the differences between each of the cases regarding the location. Similarly to the LBM streaming, a different number of memory movements are necessary

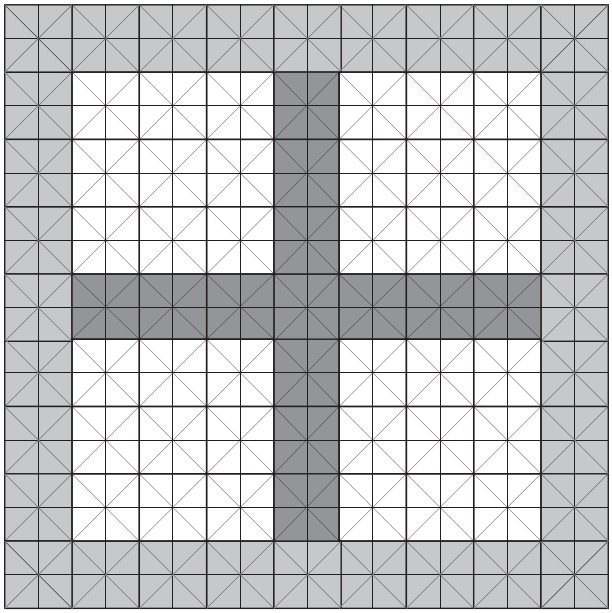

FIGURE 4 A simple scheme for our lattice-Boltzmann method approach composed by 4 *fluid blocks* (CUDA blocks) composed by ghost cells (dark-gray background), boundary (light-gray background), and fluid (white background) units

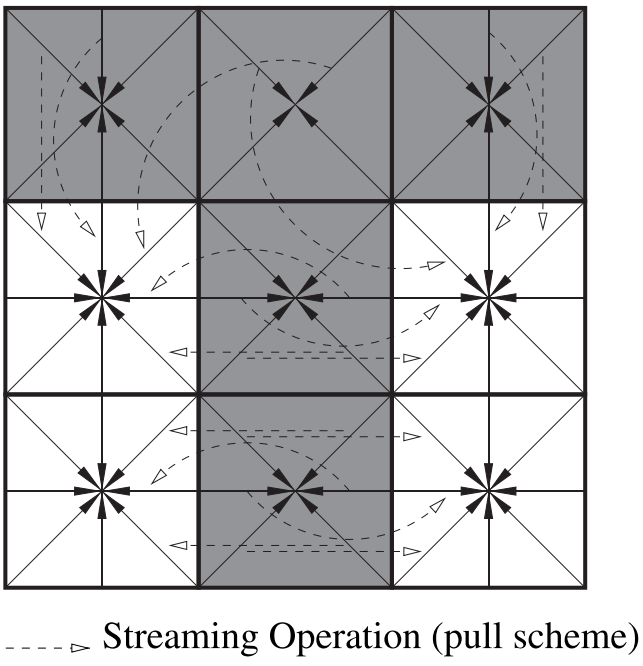

FIGURE 5 Streaming operation from *ghost cells* to fluid units

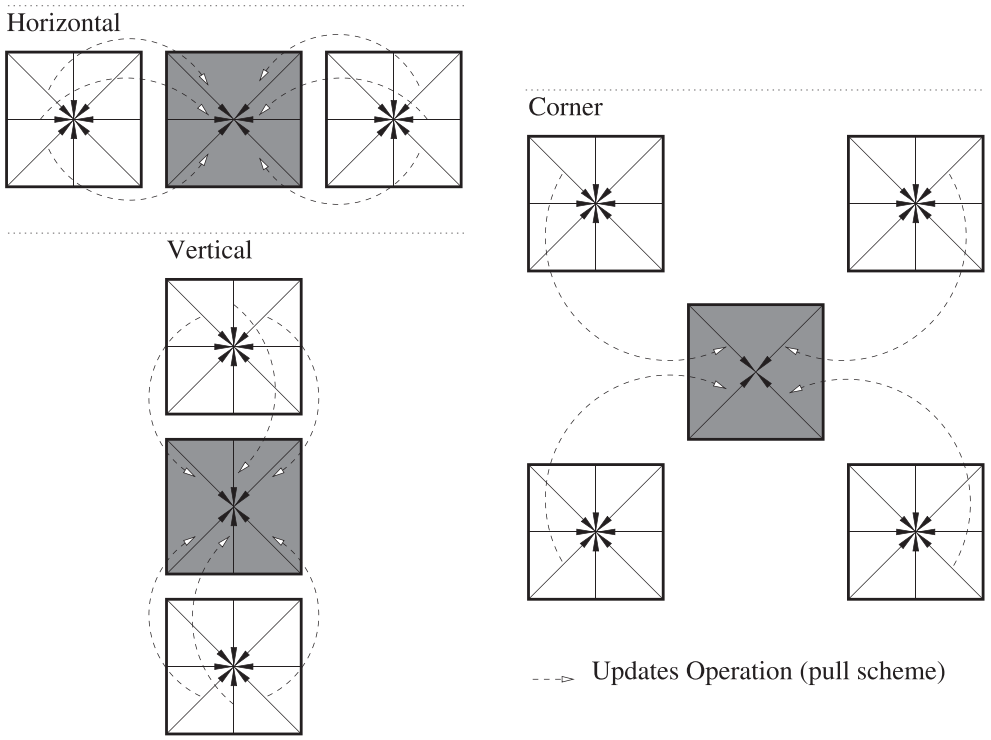

FIGURE 6 Update operation from fluid units (white background) to *ghost cells* (gray background), depending on *ghost cells* position

depending on the position of the *ghost cells*. In particular, if one *ghost cell* is located in one of the *ghost cell* rows or columns (vertical and horizontal cases in Figure 6), this needs to take 6 lattice speed from 2 different fluid units (3 lattice speed per fluid unit). However, if one *ghost cell* is positioned in one of the corners (corner case in Figure 6), then this *ghost cell* requires 4 lattice speed from 4 fluid units.

Unlike the *LBM-Standard* implementation (*pull* approach) on GPU, the CUDA blocks need to be synchronized before computing collision. This is possible using *__syncthreads()* (see Algorithm 1). The synchronizations and *ghost cells* make possible the absence of race conditions.

It is well known that the memory management has an impressive impact on performance, in particular, on those parallel computers that suffer from a high latency, such as NVIDIA GPUs or Intel Xeon Phi.[7] Furthermore, LBM is a memory-bound algorithm, so that another important optimization problem is to maximize data locality.

The previous thread-data distribution shown in Figure 3 does not exploit coalescence (contiguous threads access to continuous memory locations), when dealing with *ghost cells*, so we proposed a new memory mapping, which fits better our particular data structure. We follow the same aforementioned strategy (*structure of array*), adapting it to our approach based on *ghost cells*. Instead of mapping every lattice speed in consecutive memory locations for the whole simulation domain (Figure 2), we map the set of lattice speed of every bi-dimensional CUDA (*fluid*) block in consecutive memory locations, as graphically illustrated by Figure 7.

4 | LBM-SWAP

In this section, we explore other strategy to minimize the memory requirements for LBM simulations on NVIDIA GPUs. Unlike the previous strategy, this approach (*LBM-Swap*) does not need more memory (*ghost cells*) or change the data layout. This makes much easier the implementation and the integration with the CPU for heterogeneous implementations.[7,27,34-36] The *LBM-Swap* algorithm only needs one lattice-speed data space. For the sake of clarity and make easier the understanding in the rest of this section, let us define the *opposite* relation as follows:

$$c_{opposite(i)} = -c_i. \tag{7}$$

The *LBM-Swap* consists of swapping the lattice speed after computing the 2-main LBM steps, collide and streaming. In this way, we avoid race conditions among neighbor lattice. The *swap* function can be implemented as Pseudocode 2 describes.

Basically, this approach is based on the next property:

$$f_i\left(x + e_i\Delta t, t + \Delta t\right) \leftarrow f_{opposite(i)}(x, t), \tag{8}$$

which is symmetric and then it can be reverted by using the property $i = opposite(opposite(i))$ and Equation 7 obtaining

$$f_{opposite(i)}(x, t + \Delta t) \leftarrow f_i\left(x + e_i\Delta t, t\right). \tag{9}$$

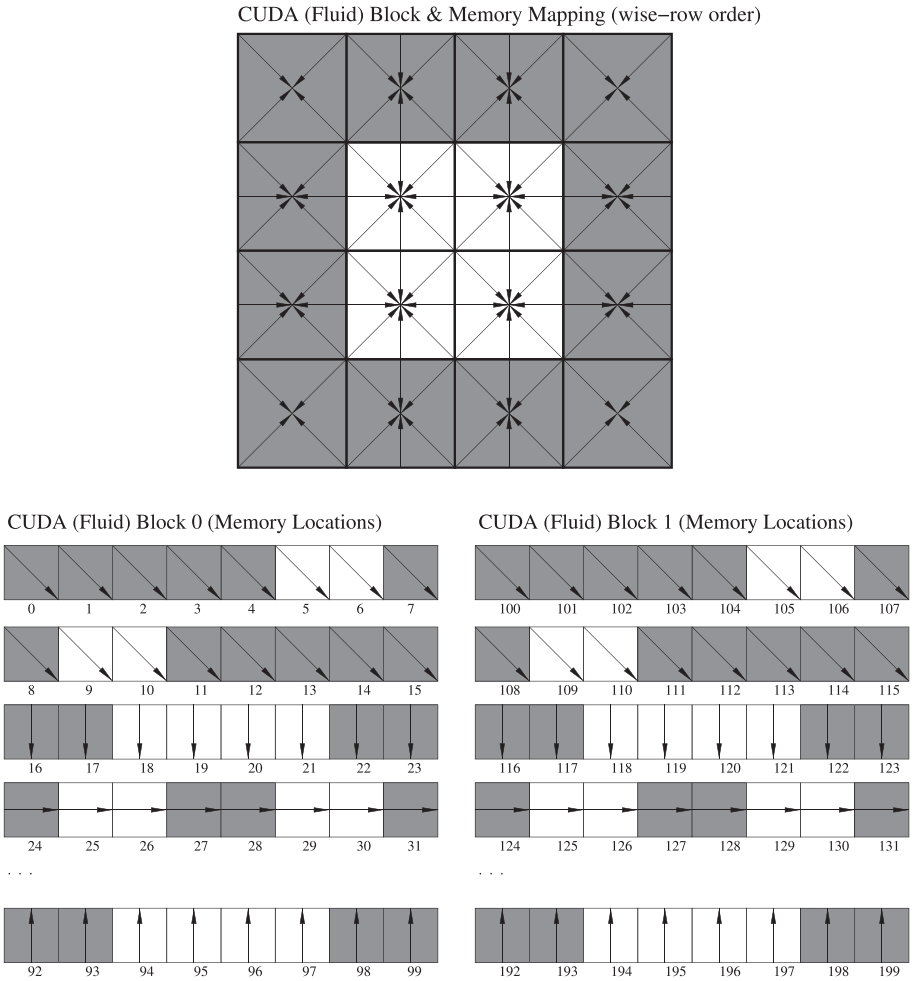

FIGURE 7 Memory and CUDA block mapping for the *1 lattice + ghost* approach

Algorithm 2 Swap implementation.

```
1: void swap(double * a, double * b){
2: double tmp = a;
3: a = b;
4: b = tmp;
5: }
```

As the *LBM-Ghost*, here we need 2 kernels, one LBM collision and one for LBM streaming. Due to GPU programming and architecture, it is necessary a strong point of synchronism among both steps to guarantee the absence of race conditions among them. This is because of the use of one lattice (*AA scheme*) instead of 2-lattice (*AB scheme*). In each kernel, we have as many threads as number of lattice (fluid) nodes. We use the same CUDA thread and memory mapping used for the *LBM-Standard* approach (Figures 2 and 3).

Pseudocode 3 describes the first kernel of the *LBM-Swap*. As shown, apart of using one lattice space (*f* in Pseudocode 3), the only difference with respect to the *LBM-Standad* consists of computing a swap operation after *LBM collision* on all lattice nodes.

The second kernel is implemented as Pseudocode 4 describes. Basically, it consists of computing streaming and swap, as described in Equation 9.

For the sake of clarity, Figure 8 graphically illustrates the swapping carried out in both LBM steps, collision (top) and streaming (bottom).

5 | PERFORMANCE EVALUATION

To carry out the experiments we have used one NVIDIA Kepler (K20c) GPU with 2496 CUDA cores at 706 Mhz and 5 GB GDDR5 of memory. More details about the specific architecture that have been used for performance evaluation are given in Table 1. The memory hierarchy of the GPU has been configured as 16 KB shared memory and 48 KB L1, since our codes cannot take advantages form a bigger shared memory. We have considered the most appropriate size of fluid block for each of the tests.

Algorithm 3 LBM-swap, kernel collision.

1: $ind = threaIdx$
2: **for** $i = 0 \rightarrow 8$ **do**
3: $\rho += f[i][ind]$
4: $u_x += c_x[i] \cdot f[i][ind]$
5: $u_y += c_y[i] \cdot f[i][ind]$
6: **end for**
7: $u_x = u_x / \rho$
8: $u_y = u_y / \rho$
9: **for** $i = 0 \rightarrow 8$ **do**
10: $cu = c_x[i] \cdot u_x + c_y[i] \cdot u_y$
11: $f_{eq} = \omega[i] \cdot \rho \cdot (1 + 3 \cdot cu + cu^2 - 1.5 \cdot (u_x)^2 + u_y)^2$
12: $f[i][ind] = f[i][ind] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
13: **end for**
14: **Swapping**
15: **for** $i = 0 \rightarrow 4$ **do**
16: $swap(f[i][ind], f[i+4][ind])$
17: **end for**

Algorithm 4 LBM-swap, kernel streaming.

1: $ind = threaIdx$
2: **for** $i = 0 \rightarrow 4$ **do**
3: $x_{stream} = x + c_x[i]$
4: $y_{stream} = y + c_y[i]$
5: $ind_{stream} = y_{stream} \cdot Nx + x_{stream}$
6: **Streaming and Swapping**
7: $swap(f[i+4][ind], f[i][ind_{stream}])$
8: **end for**

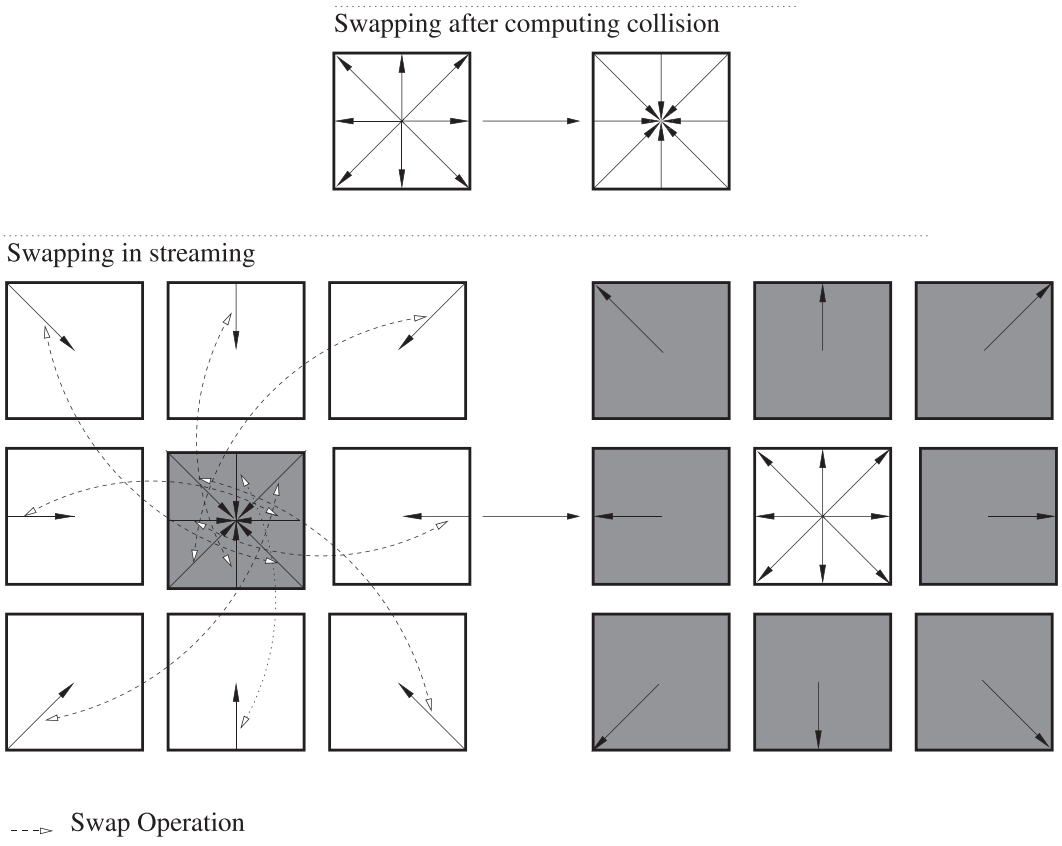

FIGURE 8 Swap operation in lattice-Boltzmann method collision (top) and in the lattice-Boltzmann method streaming (bottom)

TABLE 1 Summary of the main features of the platforms used

Platform	NVIDIA GPU Kepler K20c
Model	Kepler K20c
Frequency	0.706
Cores	2496
On-chip Mem.	SM 16/48 KB (per MP)
	L1 48/16 KB (per MP)
	L2 768 KB (unified)
Memory	5 GB GDDR5
Bandwidth	208 GB/s
Compiler	nvcc 6.0.67
Compiler Flag	-O3 -arch = sm 35

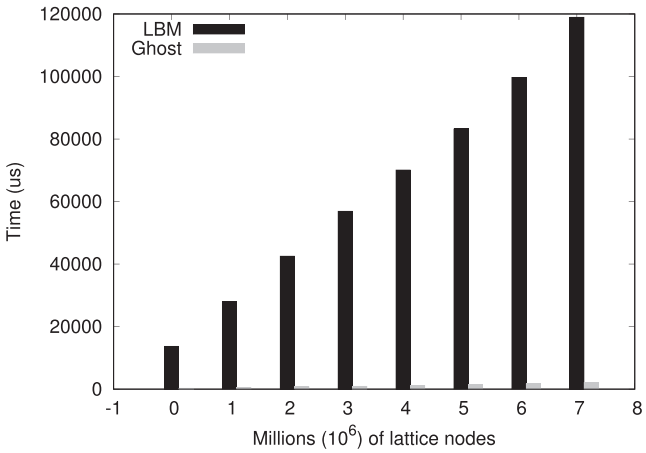

FIGURE 9 Execution time for the *LBM-Ghost* approach. LBM indicates lattice-Boltzmann method

Big fluid domains (from 45 millions of nodes) cannot be fully stored in global memory, which forces us to execute our problem in 2 steps, when using *LBM-Standard*, requiring additional memory transfers. In this case, several sub-domains must be transfered from GPU to CPU and vice versa every temporal iteration, causing a big fall in performance. Otherwise, the *LBM-Ghost* is able to achieve a better performance when dealing with big problems. Although this approach is in need of 2 kernels, instead of 1 as in the *LBM-Standard*, the time required by the new kernel (*Ghost* in Figure 9), which is in charge of updating the information in the *ghost elements*, does not cause a significant overhead. Indeed, the time consumed is less than 2% with respect to the total consumed time.

The main motivation of this work consists of reducing the memory requirements for LBM simulations on GPUs. The reduction achieved by *LBM-Ghost* represents about the 55% of the memory consumed by the *LBM-Standard*. On the other hand, the *LBM-Swap* is in need of the lowest memory requirements with respect to the other 2 implementations, needing the half of the memory required by the *LBM-Standard* and about a 5% less than the *LBM-Ghost*. Using both approaches, bigger simulations can be computed without additional and computationally expensive memory transfers.

Most of the LBM studies include the Millions of Fluid Lattice Updates Per Second (MFLUPS) ratio as a metric. As a reference, we also estimate the ideal MFLUPS[37] regarding our platform (K20c):

$$MFLUPS_{ideal} = \frac{B \times 10^9}{10^6 \times n \times 6 \times 8}, \tag{10}$$

where $B \times 10^9$ is the memory bandwidth (GB/s), n depends on LBM model (DxQn), for our framework $n = 9$, D2Q9. The factor 6 is for the memory accesses, 3 read and write operations in the streaming step and 3 read and write operations in the collision step, and the factor 8 is for double precision (8 bytes).

Figure 10 illustrates the MFLUPS achieved by all the implementations tested and an estimation for the ideal MFLUPS for our platform. The *LBM-Standard* approach is close to ideal performance for "small" problems (until 36 millions of fluid units), being the *LBM-Ghost* almost a 10% slower, because of a more complex implementation. The *LBM-Swap* is positioned as the slowest implementation tested. This implementation is about a 30% and 40% slower than the *LBM-Ghost* and *LBM-Standard*, respectively. This is mainly because of the swap operation carried out at the end of the collision kernel (Pseudocode 3).

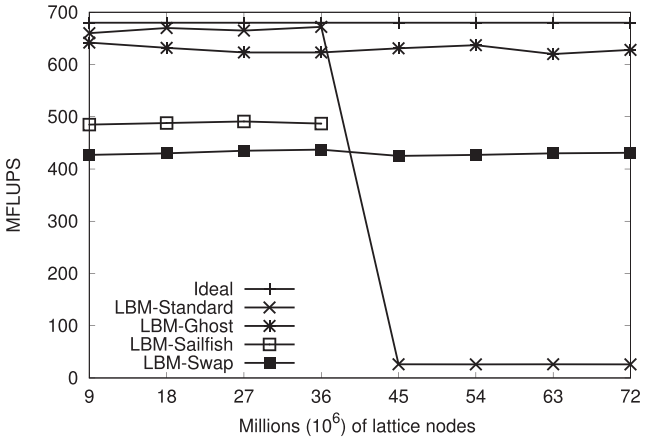

FIGURE 10 MFLUPS reached by each of the approaches. MFLUPS indicates Millions of Fluid Lattice Updates Per Second; LBM, lattice-Boltzmann method

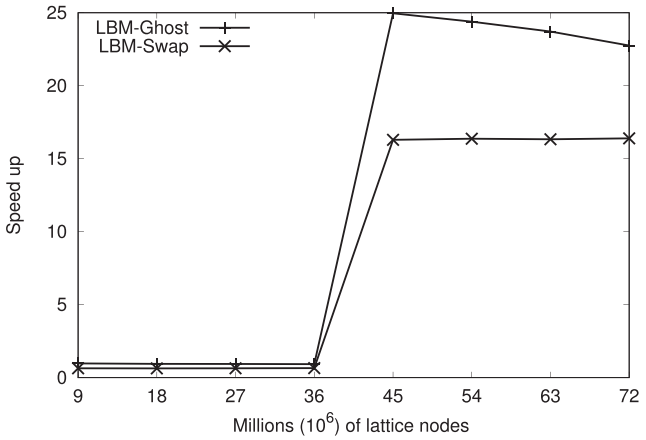

FIGURE 11 Speedup achieved by the *LBM-Ghost* and *LBM-Swap* on the *LBM-Standard*. LBM indicates lattice-Boltzmann method

However, when bigger domains are considered (from 45 to 72 millions of fluid units), the *LBM-Standard* turns out to be very inefficient, causing an important fall in performance. In contract, the performance achieved by the other 2 approaches, *LBM-Ghost* and *LBM-Swap*, keeps constant for the rest of tests. Also, as reference, we included the performance achieved by the GPU-based implementation provided in the sailfish package,[8] which is slower than *LBM-Standard* and *LBM-Ghost* and faster than *LBM-Swap* for small simulations.

Figure 11 illustrates the speedup, in terms of MFLUPS, achieved by the *LBM-Ghost* and *LBM-Swap* implementations over the *LBM-Standard*. Both approaches (*LBM-Ghost* and *LBM-Swap*) are slower than the *LBM-Standard* counterpart when executing simulations equal or smaller than 36 millions of fluid units; however, the *LBM-Standard* turns to be the slowest when dealing with bigger domains. The *LBM-Ghost* is able to achieve a peak speedup equal to 25, while the peak speedup achieved by the *LBM-Swap* is about 16.

6 | CONCLUSIONS

The limitation found in the memory capacity of GPUs and the amount of memory demanded by LBM suppose an important drawback when dealing with large problems. This work presents 2 new alternatives, *LBM-Ghost* and *LBM-Swap*, which reduce the memory used and keep a high performance for large simulations. It was carried out a detailed performance analysis in terms of time, memory requirements, speedup, and MFLUPS ratio. The implementation proposed is thoroughly detailed.

Although the *LBM-Ghost* achieves a high performance when dealing with big simulations, it makes use of nontrivial optimizations, which make difficult its implementation. Otherwise, the *LBM-Swap* is straightforward. It basically consists of swapping the lattice units of each fluid node after computing LBM collision and LBM streaming.

Also, it is important to note the *LBM-Ghost* is in need of a different data layout, which may suppose additional overheads (not considered in this work) regarding pre/post-processing to adapt the standard data layout to/from the particular data layout used by this approach. On the other hand, the *LBM-Swap* is not in need of a different data layout with respect to the *LBM-Standard*, so that no pre/post-processing is needed.

ACKNOWLEDGMENTS

This project was funded by the Spanish Ministry of Economy and Competitiveness (MINECO): BCAM Severo Ochoa accreditation SEV-2013-0323, MTM2013-40824, Computación de Altas Prestaciones-VIITIN2015-65316-P, by the Basque Excellence Research Center (BERC 2014-2017) program by the Basque Government, and by the Departament d' Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d' Execució Paral·lels (2014-SGR-1051). We also thank the support of the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT) and NVIDIA GPU Research Center program for the provided resources, as well as the support of NVIDIA through the BSC/UPC NVIDIA GPU Center of Excellence.

REFERENCES

1. Valero-Lara P, Pinelli A, Favier J, Matias MP. Block tridiagonal solvers on heterogeneous architectures. In: Inproceedings of the 2012 IEEE 10Th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12. IEEE Computer Society; 2012; Washington, DC, USA:609-616.
2. Valero-Lara P, Pinelli A, Prieto-Matias M. Fast finite difference poisson solvers on heterogeneous architectures. *Comput Phys Commun*. 2014;185(4):1265-1272.
3. Succi S. The lattice Boltzmann equation for fluid dynamics and beyond (numerical mathematics and scientific computation). *Numerical mathematics and scientific computation*. USA: Oxford university press; August 2001.
4. Bernaschi M, Fatica M, Melchiona S, Succi S, Kaxiras E. A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency Computa Pract Exper*. 2010;22:1-14.
5. Rinaldi PR, Dari EA, Vénere MJ, Clausse A. A lattice-Boltzmann solver for 3d fluid simulation on {GPU}. *Simulation Modelling Practice and Theory*. 2012;25(0):163-171.
6. Pohl T, Kowarchik M, Wilke J, Rüde U, Iglbergerm K. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. *Parallel Process Lett*. 2003;13(4):549-560.
7. Valero-Lara P, Igual FD, Prieto-Matías M, Pinelli A, Favier J. Accelerating fluid–solid simulations (lattice-Boltzmann & immersed-boundary) on heterogeneous architectures. *J Comput Sci*. 2015;10:249-261.
8. Januszewski M, Kostur M. Sailfish a flexible multi-GPU implementation of the lattice Boltzmann method. *Comput Phys Commun*. September 2014;185(9):2350-2368.
9. LBM-HPC. Last Access on 26-04-2016 to. http://www.bcamath.org/en/research/lines/CFDCT/software
10. Obrecht C, Kuznik F, Tourancheau B, Roux J-J. Scalable lattice Boltzmann solvers for {CUDA} {GPU} clusters. *Parallel Comput*. 2013;39(6-7):259-270.
11. Next Generation of CFD XFlow. Last access on 26-04-2016 to http://www.xflowcfd.com/
12. CFD Complex Physics Palabos. Last access on 26-04-2016 to http://www.palabos.org/
13. Yang W, Li K, Mo Z, Li K. Performance optimization using partitioned SPMV on GPUs and multicore CPUs. *IEEE Trans Comput*. Sept 2015;64(9):2623-2636.
14. Li K, Yang W, Li K. Performance analysis and optimization for SPMV on GPU using probabilistic modeling. *IEEE Trans Parallel and Distrib Sys*. January 2015;26(1):196-205.
15. Li K, Yang W, Li K. A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations. *IEEE Trans Parallel and Distrib Sys*. October 2016;27(10):2795-2808.
16. Yang W, Li K, Li K. A hybrid computing method of SPMV on CPU–GPU heterogeneous computing systems. *J Parallel and Distrib Comput*. 2017;104:49-60.
17. Ye Y, Li K, Wang Y, Succi T. Parallel computation of entropic lattice Boltzmann method on hybrid CPU–GPU accelerated system. *Comput Fluids*. 2013;110:114-121. 2015 parCFD.
18. Valero-Lara P. Leveraging the performance of LBM-HPC for large sizes on GPUs using ghost cells. In: In Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, December 14-16, 2016. Proceedings; 2016; Granada, Spain:417-430.
19. Carretero J, Blas JG, Ko RKL, Mueller P, Nakano K, editors. Algorithms and architectures for parallel processing. In: 16th International Conference, ICA3PP 2016, Granada, Spain, 14-16 2016 Proceedings, volume 10048 of Lecture Notes in Computer Science. Springer:2016.
20. Latt J. Technical report: how to implement your ddqq dynamics with only q variables per node (instead of 2q). *In Tufts University*. 2007:1-8.
21. Wendt JF, Anderson JD. *Computational Fluid Dynamics: An Introduction*: Springer; 2008.
22. Mohamad AA. *The Lattice Boltzmann Method—Fundamental and Engineering Applications with Computer Codes*: Springer; 2011.
23. Axner L, Hoekstra AG, Jeays A, Lawford P, Hose R, Sloot PMA. Simulations of time harmonic blood flow in the mesenteric artery: comparing finite element and lattice Boltzmann methods. *BioMed Eng OnLine*. 2000.
24. Kollmannsberger S, Geller S, Düster A, et al. Fixed-grid fluid–structure interaction in two dimensions based on a partitioned lattice Boltzmann and p-FEM approach. *Int J Numer Methods Eng*. 2009;79(7):817-845.
25. Malaspinas O, Sagaut P. Consistent subgrid scale modelling for lattice Boltzmann methods. *J Fluid Mech*. 2012;700:514-542.
26. Marié S, Ricot D, Sagaut P. Comparison between lattice Boltzmann method and Navier-Stokes high order schemes for computational aeroacoustics. *J Comput Phys*. March 2009;228(4):1056-1070.
27. Valero-Lara P, Pinelli A, Prieto-Matias M. Accelerating solid-fluid interaction using lattice-Boltzmann and immersed boundary coupled simulations on heterogeneous platforms. *Procedia Computer Science*. 2014;29(0):50-61. 2014 International Conference on Computational Science.
28. He X, Luo L-S. A priori derivation of the lattice Boltzmann equation. *Phys Rev E*. Jun 1997;55:R6333-R6336.
29. Qian YH, D'Humières D, Lallemand P. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*. 1992;17(6):479.
30. Gross E, Bhatnagar P, Krook M. A model for collision processes in gases. i: small amplitude processes in charged and neutral one-component system. *Phys Rev E*. 1954;94:511-525.
31. Wellein G, Zeiser T, Hager G, Donath S. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*. 2006;35(89):910-919. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

32. Bernaschi M, Fatica M, Melchionna S, Succi S, Kaxiras E. A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice and Experience*. 2010;22(1):1-14.
33. Valero-Lara P, Jansson J. LBM-HPC—an open-source tool for fluid simulations. Case study unified parallel C (UPC-PGAS). *In 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015*. Chicago, IL, USA, September 8-11, 2015; 2015:318-321.
34. Valero-Lara P. Accelerating solid–fluid interaction based on the immersed boundary method on multicore and GPU architectures. *J Supercomput*. 2014:1-17.
35. Valero-Lara P. *A fast multi-domain lattice-Boltzmann solver on heterogeneous (multicore-GPU) architectures. 14th International Conference Computational and Mathematical Methods in Science and Engineering*. 2014;4:1239-1250.
36. Valero-Lara P, Jansson J. Heterogeneous CPU+ GPU approaches for mesh refinement over lattice-Boltzmann simulations. *Concurrency and Computation: Practice and Experience*. 2016.
37. Shet AG, Sorathiya SH, Krithivasan S, et al. Data structure and movement for lattice-based simulations. *Phys Rev E*. July 2013;013314:88.

How to cite this article: Valero-Lara P. Reducing memory requirements for large size LBM simulations on GPUs . *Concurrency Computat: Pract Exper*. 2017;29:e4221. https://doi.org/10.1002/cpe.4221