Accelerating fluid-solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures☆

Pedro Valero-Lara[a,b], Francisco D. Igual[b], Manuel Prieto-Matías[b], Alfredo Pinelli[c]

[a] *BCAM-Basque Center for Applied Mathematics, Bilbao, Spain.*
[b] *Dept. Arquitectura de Computadores y Automaática, Universidad Complutense de Madrid, Madrid, Spain.*
[c] *School of Engineering and Mathematical Sciences, City University London, London, United Kingdom.*

Abstract

We propose a numerical approach based on the Lattice-Boltzmann (LBM) and Immersed Boundary (IB) methods to tackle the problem of the interaction of solids with an incompressible fluid flow. We explain in detail the parallelization of the complete method on two different heterogeneous platforms based on massively parallel hardware accelerators: NVIDIA Kepler GPU and Intel Xeon Phi. we describe a number of software optimizations, mainly focusing on memory management and host-accelerator communication. Based on a thorough perforrmance evaluation, we demonstrate that the baseline LBM implementation achieves satisfactory results on hardware accelerators. Unfortunately, when coupling LBM and IB methods on the heterogeneous platforms, the overheads of IB degrade the overall performance. As an alternative, we explore different heterogeneous implementations that effectively hide such overheads and allow us to exploit both the multi-core and the hardware accelerator in a co-operative way.

Keywords: Parallel Computing, Computational Fluid Dynamics, Fluid-Solid Iteraction, Lattice-Boltzmann Method, Immersed-Boundary Method, Heterogeneous Computing

1. Introduction

The dynamics of a solid in a flow field is a research topic currently with a growing interest in many scientific communities. It is intrinsically interdisciplinary (structural mechanics, fluid mechanics, applied mathematics, ...) and covers a broad range of applications (e.g. aeronautics, civil engineering, biological flows, etc.) The number of works in this field reflects the growing importance of the study of the dynamics in the solid-fluid interaction [14, 15, 16, 17]. The use of accelerated, heterogeneous architectures to compute the fluid field is widely used within the Computational Fluid Dynamics (CFD) community due to the computational requirements of such applications, with significant performance results [11, 12, 13]. In contrast, the solid-fluid interaction has only recently gained wider interest.

The main objective of this work is the reduction the overhead caused by the simulation of solid-fluid interaction on heterogeneous platforms. In particular, we propose a number of hybrid strategies that distribute parts of the computation either to the hardware accelerator or the general-purpose CPU, depending on how they adapt to each part of the solver. We also give notes on specific optimizations to tune each one of the parts of the computation, adapting them to the characteristics of each architecture.

Classical fluid solvers based on the unsteady incompressible Navier Stokes equations may turn out to be inefficient or difficult to tune to achieve maximum performance on new massively-parallel platforms [18, 19]. A choice that better meets their characteristics is based on modeling the fluid flow through the Lattice Boltzmann method (LBM). Several recent works have shown that the combination of hardware accelerators and methods based on the LBM algorithm can achieve impressive performances due to the intrinsic characteristics of the algorithm. Certainly, the computing stages of LBM are amenable to fine grain paralelization in an almost straightforward way (see, for example, [11, 12] and references therein). Nevertheless, few works extend the parallel efficiency of LBM to cases involving geometries bounded by complex, moving or deformable

boundaries. A very recent work that covers a subject closely related with the present contribution is the one by [13] where a new efficient 2D implementation of LBM method for fluids flowing in geometries with curved boundary using GPUs platforms is proposed. Curved boundaries are taken into account via a non equilibrium extrapolation scheme developed by [21]. S. K. Laytona et al. [31] propose a GPU implementation based on the *IB projection* method [32], which computes the influence of one rigid solid into Navier-Stokes solver through the use of sparse linear algebra routines, using the open-source `Cusp` library to carry out these routines. This approach requires a higher computational compared with ours, attaining similar accuracy.

We consider a different approach for computing the influence of a solid immersed in a incompressible flow, which is based on *different forcing* approach [6] able to deal with complex, moving or deformable boundaries. It has been used on Lattice-Boltzmann [29] and Navier-Stokes [7] based solvers. In [20], authors proposed the use of heterogeneous CPU-GPU platforms to minimize the overhead that the computing of the solid presence exhibits. This paper extends the aforementioned contribution in different ways. Our numerical framework is based on LBM coupled with an Immersed Boundary (IB) method able to deal with complex, moving or deformable boundaries [1, 2, 3, 4, 5, 6, 7, 29]. Special emphasis is given to the algorithmic and implementation techniques adopted to keep the solver highly efficient on heterogeneous host-accelerator platforms with independent memory spaces, based both on GPUs or the recently introduced Intel Xeon Phi accelerator.

This paper is structured as follows. Section 2 introduces the physical problem at hand and the general numerical framework that has been selected to cope with it: Lattice-Boltzmann method (LBM) coupled with Immersed-Boundary (IB) technique based on the use of a set of *Lagrangian* nodes distributes along the solid boundaries. In Section 3, the specific potential parallel features of IB method over multicore and massively parallel architectures are presented. Different LBM approaches for dealing with the particular features of the architectures considered are discussed in Section 4. In Section 5, we detail the

parallel strategies envisaged to optimally enhance the performance of the global LBM-IB algorithm on CPU-GPU heterogeneous platforms. Finally, Section 6 details the performance analysis of the proposed techniques and in Section 7 some conclusions are outlined.

2. The Lattice Boltzmann and Immersed Boundary methods

The Lattice Boltzmann method combined with an Immersed Boundary technique is highly attractive when dealing with bodies for two main reasons: the shape of the boundary, tracked by a set of Lagrangian nodes is a sufficient information to impose the boundary values; and the force of the fluid on the immersed boundary is readily available and thus easily incorporated in the set of equations that govern the dynamics of the immersed object. In addition, it is also particularly well suited for massively parallelized simulations, as the time advancement is explicit and the computational stencil is formed by few local neighbors of each computational node (*support*). The fluid is discretized on the regular Cartesian mesh while the shape of the solids is discretized in a Lagrange fashion by a set of points which obviously do not necessarily coincide with mesh points. The Lattice Boltzmann method has been extensively used in the past decades (see [22] for a complete overview) and it is now regarded as a powerful and efficient alternative to classical Navier Stokes solvers. In particular, LBM has been used to simulate high Reynolds turbulent flows both using Direct Numerical Simulation and Large Eddy Simulation [23]. Another challenging application where LBM has proved to be quite successful concerns aeroacoustics problems [24]. In what follows we briefly recall the basic formulation of the method. The LBM is based on an equation that governs the evolution of a discrete distribution function $f_i(\mathbf{x}, t)$ describing the probability of finding a particle at Lattice site $\mathbf{x}$ at time t with speed $\mathbf{v} = \mathbf{e}_i$. In this work, we consider the *BGK* formulation [25] that relies upon an unique relaxation time τ toward the equilibrium distribution $f_i^{(eq)}$:

$$f_i\left(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t\right) - f_i\left(\mathbf{x}, t\right) = -\frac{\Delta t}{\tau}\left(f\left(\mathbf{x}, t\right) - f_i^{(eq)}\left(\mathbf{x}, t\right)\right) + \Delta t F_i \qquad (1)$$

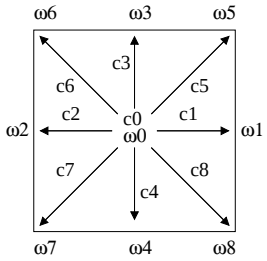

Figure 1: The standard two-dimensional 9-speed lattice (*D2Q9*) used.

The particles can move only along the links of a regular Lattice defined by the discrete speeds ($e_0 = c(0,0);\ e_i = c(\pm 1, 0), c(0, \pm 1), i = 1 \cdots 4;\ e_i = c(\pm 1, \pm 1), c(\pm 1, \pm 1), i = 5 \cdots 8$ with $c = \Delta x / \Delta t$) so that the synchronous particle displacements $\Delta \mathbf{x}_i = \mathbf{e}_i \Delta t$ never take the fluid particles away from the Lattice. For the present study, the standard two-dimensional 9-speed Lattice *D2Q9* (Figure 1) is used [26], but all the techniques that will be presented can be extended in a straightforward manner to three dimensional lattices.

The equilibrium function $f^{(eq)}(\mathbf{x}, t)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution [27]:

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

In Equation 2, c_s is the speed of sound ($c_s = 1/\sqrt{3}$) and the weight coefficients ω_i are $\omega_0 = 4/9$, $\omega_i = 1/9,\ i = 1 \cdots 4$ and $\omega_5 = 1/36,\ i = 5 \cdots 8$ according to the current normalization. The macroscopic velocity $\mathbf{u}$ in Equation 2 must satisfy a Mach number requirement $\mid \mathbf{u} \mid / c_s \approx M << 1$. This stands as the equivalent of the CFL number for classical Navier Stokes solvers. Finally, in Equation 1, F_i represents the contribution of external volume forces at lattice level that in our case include the effect of the immersed boundary. Given any external volume force $\mathbf{f}^{(ib)}(\mathbf{x}, t)$, the contribution on the lattice are computed according to the formulation proposed by [21] as:

$$F_i = \left(1 - \frac{1}{2\tau} \right) \omega_i \left[\frac{\mathbf{e_i} - \mathbf{u}}{c_s^2} + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^4} \mathbf{e_i} \right] \cdot \mathbf{f_{ib}} \tag{3}$$

The multi-scale Chapman Enskog expansion of Equation 1, neglecting terms of $O(\epsilon M^2)$ and using expression 3, returns the Navier-Stokes equations with body forces and the kinematic viscosity related to lattice scaling as $\nu = c_s^2(\tau - 1/2)\Delta t$.

Without the contribution of the external volume forces stemming from the immersed boundary treatment, Equation 1 is typically advanced in time in two stages: *collision* and *streaming*.

Given $f_i(\mathbf{x}, t)$ compute:

$\rho = \sum f_i(\mathbf{x}, t)$ and $\rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x}, t)$

Collision stage:

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

Streaming stage:

$$f_i (\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^* (\mathbf{x}, t + \Delta t)$$

Depending on the ordering of the two stages, two different strategies arise. The classical approach is known as the *push* method and performs collision before streaming. We have adopted instead for *pull* method [28], which performs the steps in the opposite order. This can lead to an important performance enhancement on fine grained parallel machines. A short discussion about the different implementations and achieved performances using the two orderings will be detailed later on.

Next, we briefly introduce the Immersed Boundary method that we use both to enforce boundary values and to recover the fluid force exerted on immersed objects within the framework of the LBM algorithm [6, 7]. In the taken IB approach (as in several others), the fluid is discretized on a regular Cartesian lattice while the immersed objects are discretized and tracked in a Lagrangian fashion by a set of markers distributed along their boundaries. The general setup of the present Lattice Boltzmann-Immersed Boundary method can be recast in the following algorithmic sketch.

Given $f_i(\mathbf{x}, t)$ compute:

$\rho = \sum f_i(\mathbf{x}, t)$ and

$\rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x}, t) + \frac{\Delta t}{2}\mathbf{f}_{ib}$

Collision stage:

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

Streaming stage:

$\hat{f}_i (\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = \hat{f}_i^* (\mathbf{x}, t + \Delta t)$

Compute :

$\hat{\rho} = \sum \hat{f}_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t)$ and
$\hat{\rho}\hat{\mathbf{u}} = \sum \mathbf{e}_i \hat{f}_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t)$

Interpolate on Lagrangian markers (volume force):

$\hat{\mathbf{U}}(\mathbf{X}_k, t + \Delta t) = \mathcal{I}(\hat{\mathbf{u}})$ and
$\mathbf{f}_{ib}(\mathbf{x}, t) = \frac{1}{\Delta t} \mathcal{S} \left(\mathbf{U}_d(\mathbf{X}_k, t + \Delta t) - \hat{\mathbf{U}}(\mathbf{X}_k, t + \Delta t) \right)$

Repeat collision with body forces (see 3) and Streaming:

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

$f_i (\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^* (\mathbf{x}, t + \Delta t)$

As outlined above, the basic idea consists in performing each time step twice. The first one, performed without body forces, allows to predict the velocity values at the immersed boundary markers and the force distribution that restores the desired velocity boundary values at their locations. The second one applies the regularized set of singular forces and repeats the procedure advancing (using Equation 3) to determine the final values of the distribution function at the next time step. The key aspects of the algorithm and of its efficient implementation depend on the way the interpolation $\mathcal{I}$ and the $\mathcal{S}$ operators (termed as spread from now on) are applied. Here, following [6] and [7] we perform both operations (interpolation and spread) through a convolution with a compact support mollifier meant to mimic the action of a Dirac's delta. Combining the two operators we can write in a compact form:

$$\mathbf{f}_{(ib)}(\mathbf{x}, t) = \frac{1}{\Delta t} \int_\Gamma \left(\mathbf{U_d}(\mathbf{s}, t + \Delta t) - \int_\Omega \hat{\mathbf{u}}(\mathbf{y}) \tilde{\delta}(\mathbf{y} - \mathbf{s}) d\mathbf{y} \right) \tilde{\delta}(\mathbf{x} - \mathbf{s}) d\mathbf{s} \qquad (4)$$

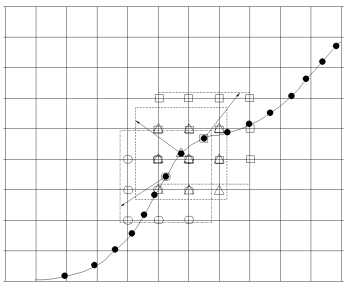

Figure 2: An immersed curve discretized with *Lagrangian* points •. Three consecutive points are considered with the respective supports.

where $\tilde{\delta}$ is the mollifier, to be defined later, Γ is the immersed boundary, Ω is the computational domain, and $\mathbf{U_d}$ is the desired value on the boundary at the next time step. The discrete equivalent of 4 is simply obtained by any standard composite quadrature rule applied on the union of the supports associated to each Lagrangian marker. As an example, the quadrature needed to obtain the force distribution on the lattice nodes is given by:

$$f^l_{ib}(x_i, y_j) = \sum_{n=1}^{N_e} F^l_{ib}(\mathbf{X_n})\tilde{\delta}(x_i - X_n, y_j - Y_n)\epsilon_n \tag{5}$$

where the superscript l refers to the l^{th} component of the immersed boundary force, (x_i, y_j) are the lattice nodes (*Cartesian* points) falling within the union of all the supports, N_e is the number of Lagrangian markers and ϵ_n is a value to be determined to enforce consistency between interpolation and the convolution 5. More details about the method and in particular about the determination of the ϵ_n values can be found in [7].

In what follows we will give more details on the construction of the support cages surrounding each Lagrangian marker since it plays a key role in the parallel implementation of the IB algorithm. Figure 2 illuastrates an example of the portion of the lattice units that falls within the union of all supports. As already mentioned, the embedded boundary curve is discretized into a number of markers $\mathbf{X}_I, \quad I = 1..N_e$. Around each marker $\mathbf{X}_I$ we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes

of the underlying Eulerian lattice for each direction; (ii) the number of nodes of the lattice contained in the cage must be minimized. The modified kernel, obtained as a cartesian product of the one dimensional function [8]

$$\tilde{\delta}(r) = \begin{cases} \frac{1}{6}\left(5 - 3|r| - \sqrt{-3(1-|r|)^2+1}\right) & \\ & 0.5 \leq |r| \leq 1.5 \\ \frac{1}{3}\left(1 + \sqrt{-3r^2+1}\right) & 0.5|r| \leq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

will be identically zero outside the square Ω_I. We take the edges of the square to measure slighlty more than three lattice spacings Δ (i.e., the edge size is $3\Delta + \eta = 3 + \eta$ in the actual LBM normalization). With such choice, at least three nodes of the lattice in each direction fall within the cage. Moreover a value of $\eta << 1$ ensures that the mollifier evaluated at all the nine (in two dimensions) lattice nodes takes on a non zero value. The interpolation stage is performed locally on each nine points support: the values of velocity at the nodes withing the support cage centered about each Lagrangian marker deliver approximate values (i.e., second order) of velocity at the marker location. The force spreading step requires information from all the markers, typically spaced $\Delta = 1$ apart along the immersed boundary. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring markers, as in 5. The outlined method has been validated for several test cases including moving rigid immersed objects and flexible ones [29].

Finally, we close this section presenting several test cases in order to validate the implementation of the code by comparing the numerical results obtained with other studies. One of the classical problems in Computational Fluid Dynamic is the determination of the two-dimensional incompressible flow field around a circular cylinder, which is a fundamental problem in engineering applications. Several Reynolds numbers (20, 40 and 100) have been tested with the same configuration. The cylinder diameter D is equal to 40. The flow space is composed by a mesh equal to $40D$ (1600) $\times$ $15D$ (600). The boundary con-

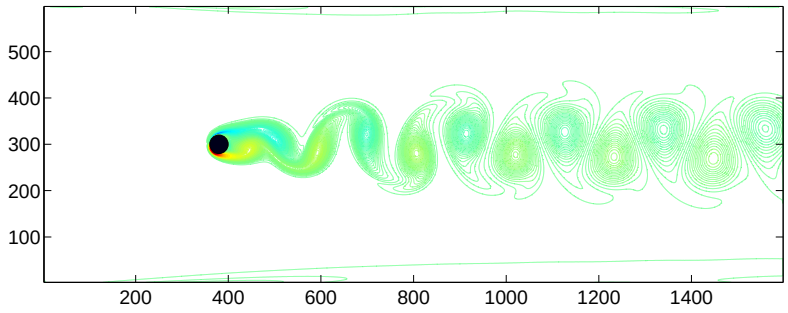

Figure 3: Vorticity with Re=100.

Author	**Re = 20**	**Re = 40**	**Re = 100**	
	C_D	C_D	C_D	C_L
Calhoun (2002)	2.19	1.62	1.33	0.298
Rusell (2003)	2.22	1.63	1.34	-
Silva (2003)	2.04	1.54	1.39	-
Xu (2006)	2.23	1.66	1.423	0.34
Zhou (2012)	2.3	1.7	1.428	0.315
This work	2.3	1.7	1.39	0.318

Table 1: Experimental results.

ditions are set as: Inlet: $\mathbf{u} = U, \mathbf{v} = 0$, Outlet: $\frac{\partial \mathbf{u}}{\partial x} = \frac{\partial \mathbf{v}}{\partial x} = 0$, Upper and lower boundaries: $\frac{\partial \mathbf{u}}{\partial y} = 0, \mathbf{v} = 0$, Cylinder surface: $\mathbf{u} = 0, \mathbf{v} = 0$, Volume fraction: 0.5236%. When Reynolds number is 20 and 40, there is no vortex structure formed during the evolution. The flow field is laminar and steady. In contrast, for the Reynolds number of 100, the symmetrical rectangular zones disappear and an asymmetric pattern is formed. The vorticity is shed behind the circular cylinder, and vortex structures are formed downstream. This phenomenon is graphically illustrated in Figure 3.

Two important dimensionless numbers are studied, the drag ($CD = \frac{F_D}{0.5\rho U^2 D}$) and lift ($CL = \frac{F_L}{0.5\rho U^2 D}$) coefficients. F_D corresponds to the resistance and F_L is the lifting force of the circular cylinder, ρ is the density of the fluid, and U is the velocity of inflow. In order to verify the numerical results, the coefficients were

calculated and compared with the results of previous studies (Table 1). The drag coefficient for Reynolds number of 20 and 40 is equal to the results presented by Zhou et al. [13]. The drag coefficient obtained for Reynolds number of 100 is identical to the results obtained by Silva et al. [17], and the lift coefficient is close to the presented by Zhou et al. [13].

3. Immersed Boundary Method Implementations

This section presents the strategy that we have adopted for the efficient parallelization of the IB algorithm when executed on CPU-GPU heterogeneous platforms. The computations related with the *Lagrange* markers (support) distributed on the solid/s surface can be parallelized efficiently on both, CPU and GPU. As already mentioned the whole algorithm can be seen as a two steps procedure: a first, global LBM update, and a subsequent local correction to impose the boundary values. It is well known that the memory management plays a crucial role in the performance. To compute the IB method, it is necessary to store the information about the coordinates, velocities and forces of all the *Lagrangian* points and their supports. A set of memory management optimizations, which depends on the access pattern, have been carried out for the IB method implementation on both platforms, multicore and GPU, to achieve an effective memory usage.

3.1. Memory management

Two different memory management approaches are proposed depending on the use of multicore or GPU, since both architectures exhibit different memory features and a substantially different hierarchy. The multicore approach stores the information of a particular *lagrangian* point and its support in nearby memory locations, which benefits the exploitation of coarse grain parallelism. In contrast, in order to achieve a coalesced access to global memory, the GPU approach distributes the information of all *lagrangian* points in a set of one-dimensional arrays. In this way, contiguous threads access to contiguous memory locations.

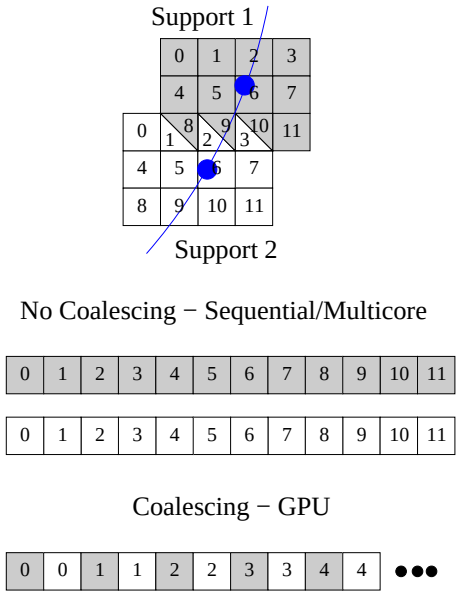

Figure 4: Memory management and mapping on sequential/multicore (left) and GPU (right) platforms for two consecutive *Lagrangian* points.

Particular attention was paid to the access pattern to the information of the supports. It was necessary to store the information of the set of all supports in a unified array to carry out coalescing memory accesses, achieving that elements which share the same index on different supports are located in continuous memory locations. For clarity, Figure 4 shows the memory mapping performed on both platforms in a simplified example with two consecutive *Lagrangian* points. As shown, the use and the exploitation of each memory hierarchy, main (CPU) and global (GPU) memory, is totally different. Furthermore, an additional advantage is found in the use of the memory structures, allowing the transfer the information of the solid(s) between both memories by carrying out a single memory transfer.

3.2. Parallelization approaches on multi-core and GPU

The degree of parallelism of the IB method is given by the number of *Lagrangian* points. The multicore approach carries out a coarse-grain parallelism

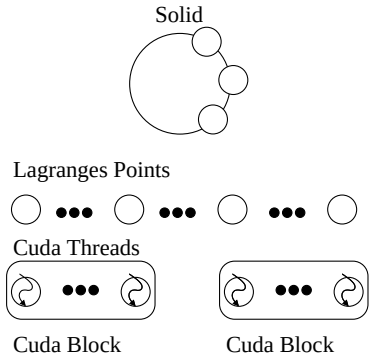

Figure 5: CUDA block-thread distribution for *Lagrangian* points.

by mapping a set of continuous *Lagrangian* points on each core which are solved sequentially. This distribution is well balanced and the use of the memory is optimized by using the memory structures previously described (Figure 4). The set of *Lagrangian* points can be easily parallelized with this approach, annotating some of its loops with OpenMP pragmas.

On the GPU, the implementation consists of using two basic kernels. The first one, denoted as Immersed Forces Computation (IFC) kernel, assembles the velocity field on the supports, undertakes the interpolation at the *Lagrangian* markers and determine the *Eulerian* volume force field on each node of the union of the supports. The second kernel, denoted as Body Forces Computation (BFC) kernel, computes the lattice forces and repeat the LBM time update only on the union of the supports including the IB forces contribution. Both kernels use the same CUDA block-thread distribution (Figure 5). The first kernel computes the whole IB method. It consists of computing these major steps:

1. *Velocities interpolation.* The input parameters of this step are loaded from global memory to local registers using coalesced memory accesses.
2. *Force computation.* The parameters are held in both, local and global memory (coalesced accesses). The computed forces are held in local registers.
3. *Spread the forces.* The parameters are used from local and global memory and the results are stored in global memory by using atomic operations.

Algorithm 1 IFC kernel.

```
1: IFC_kernel(solid s,U_x,U_y)
2: vel_x, vel_y, force_x, force_y
3: for i = 1 → numSupport do
4:     vel_x+ = interpol(U_x[s.Xsupp[i]], s)
5:     vel_y+ = interpol(U_y[s.Ysupp[i]], s)
6: end for
7: force_x = computeForce(vel_x, s)
8: force_y = computeForce(vel_y, s)
9: for i = 1 → numSupport do
10:    AddAtom(s.XForceSupp, spread(force_x, s))
11:    AddAtom(s.YForceSupp, spread(force_y, s))
12: end for
```

After the spreading step the forces are stored in the global memory by using *atomic* functions. These *atomic* functions are performed to prevent race conditions. Particularly, we used these operations to avoid incoherent executions, since the supports of different *Lagrangian* points can share the same *Eulerian* points, as graphically shown in Figure 2. The pseudo-code of the IFC kernel is graphically illustrated in Algorithm 1.

After the execution of the IB related computations (IFC kernel), the lattice forces as in equation 3 (Section 2) need to be determined. Before tackling this next stage it is necessary to introduce a synchronization point that guarantees that all the IB forces have been actually computed on all the points within the union of the supports. Nonetheless, the global memory access required to determine the system of lattice forces is larger than in the previous stage: 9 directions for each lattice node in the support. Also in this case to inhibit *race* conditions it has been necessary to resort to *atomic* functions. As for the case of the equilibrium distribution, also here the computation of the lattice force contributions is carried out using registers. The pseudo-code for this final kernel is given in algorithm 2.

Algorithm 2 BFC kernel.

1: BFC_kernel(solid s,f_x,f_y)
2: F_{body}(Body Force),x, y,vel_x, vel_y
3: **for** $i = 1 \rightarrow numSupport$ **do**
4: $x = s.Xsupport[i]$
5: $y = s.Ysupport[i]$
6: $vel_x = s.VelXsupport[i]$
7: $vel_y = s.VelYsupport[i]$
8: **for** $j = 1 \rightarrow 9$ **do**
9: $F_{body} = (1-0.5\cdot\frac{1}{\tau})\cdot w[j]\cdot(3\cdot((c_x[j]-vel_x)\cdot(f_x[x][y])+(c_y[j]-vel_y)\cdot f_y[x][y]))$
10: $AddAtom(f^{n+1}[j][x][y], F_{body})$
11: **end for**
12: **end for**

4. Lattice-Boltzmann Implementations

This section explores different strategies to enhance performance of LBM according to the architectures used in this work. Special emphasis is put on memory management and how to implement the different steps of LBM.

4.1. Parallelization approach

Falta!!

4.2. Memory mapping strategies

Being a memory-bound algorithm, it is expected that memory management has a substantial impact on the attained performance. This is even more relevant for hardware accelerator processors, usually following a many-core architectural paradigm. As an example, especially on warp-oriented architectures (e.g. GPUs), it is important that contiguous threads reach contiguous memory locations in the same cycle, thus diminishing the number of memory accesses (usually referred as *coalesced* memory accesses in the literature). Clearly, the information for each *lattice* node should be stored in sequential memory locations that reflect their geometrical ordering to improve coalescing. In contrast, other

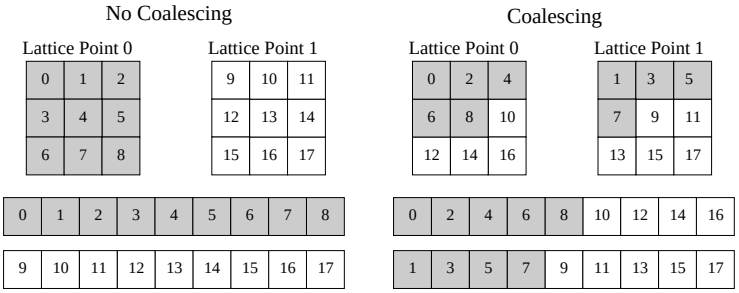

Figure 6: Uncoalesced and coalesced approach for two *lattice* points.

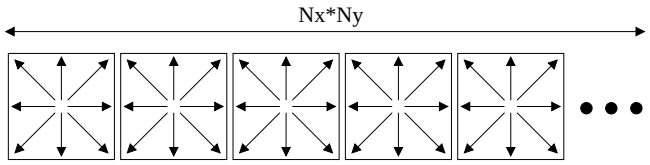

Figure 7: Uncoalesced approach.

approaches can be more efficient on other type of architectures, e.g. shared memory multi-cores.

Here, we propose a number of memory access strategies adapted to LBM, namely:

1. *Uncoalesced*. This strategy does not suppose an elaborate management of memory. Basically, the set of 9 lattice directions is stored in consecutive locations of memory. Thus, the lattice level is stored in memory as an *Array of Structures* (AoS). To better clarify the problem, Figure 6 (left) and 7 illustrate this approach. Although it can be efficiently implemented for those systems which exploit a coarse grain parallelism, this memory mapping makes it difficult for vector loads to work efficiently, due to the displacement of memory among the same lattice-direction of consecutive lattice units.
2. *Coalesced*. This alternative is given to mitigate the inconveniences found in the previous approach. **It consists of storing in a consecutive fashion each of the 9 lattice directions. (Creo que la diferencia**

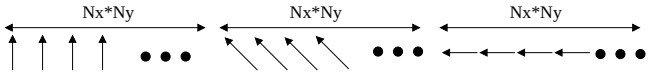

Figure 8: Coalesced approach.

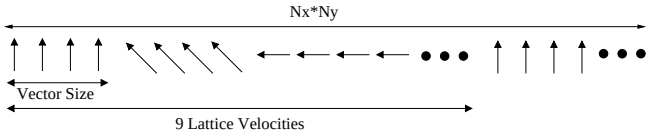

Figure 9: Hybrid approach for a vector size equal to 4.

entre este y el anterior no queda clara con estas dos frases). This approach has been widely used on CUDA-based implementations and it is considered the optimal approach for this kind of architectures [11, 12]. Henceforth, we use a different memory management based on *Structure of Arrays* (SoA). Despite this approach is more amenable to vector units, we find a large space of memory, as big as the size of the fluid domain, through the different directions (Figure 6 (right) and 8), what might not be so advantageous in other architectures such as multicore or Intel Xeon Phi due to the evident loss of locality, and thus the inability of efficiently use the cache memories.

3. *Hybrid.* This approach arises to take advantage of the main features of the aforementioned strategies. It consists of joining both features, a small distance among the 9 lattice-direction and a efficient exploitation of vector units. In this sense, groups of the same lattice-direction are located in memory consecutively (Figure 9). The vector size usually imposes the maximum number of elements to be grouped. **Faltan las ventajas especficas de este approach con respecto a lo anterior).**

To exploit the high data parallelism presented in LBM, we will keep in memory two copies of the lattice (f_1 f_2) in all implementations. Each time step alternatively takes one of the copies as the input, and writes the result to the second.

4.3. Push and pull implementations

Depending on the ordering of the collision and streaming stages two different strategies arise. The classical approach is known as *push* method and performs collide before streaming. Otherwise, the *pull* approach performs the steps in the opposite order.

4.3.1. The pull approach

The *pull* computational scheduling of LBM was introduced by [28] and has been recently consider by [12]. This is an efficient approach based on a single-loop strategy. Each lattice node can be independently computed by performing one complete time step of LBM. In general, the *pull* method reorganizes the memory pattern access by changing the ordering of the LBM steps:

1. Move distribution functions $f_i(x + c_i\Delta t, t + \Delta t)$ values from main/global memory to local memory and perform streaming.
2. Compute the macroscopic averages ρ, u (local memory).
3. Calculate the collision step $f_i^{(eq)}$ (local memory).
4. Copy the new values f_i into main/global memory.

A schematical sketch of the LBM implementation is given in Algorithm 3.

Basically, this strategy does not need any synchronization among steps, being very profitable for parallel architectures. Furthermore, the top regions of the memory hierarchy can be efficiently exploited, as the macroscopic level can be completely computed without transfers to/from main memory, better exploiting cache re-use.

4.3.2. The push approach

Next, we introduce the main characteristics of the LBM implementation based on *push* approach (collide-stream strategy). This computational scheduling of LBM has been used in numerous previous works [13, 30, 11]. In general, the *push* method divides the LBM steps into two routines: the first one computes the collide and stream steps; the second one computes the macroscopic

Algorithm 3 LBM *pull*.

Pull (f_1,f_2,ϖ,c_x,c_y)

x, y

x_{stream}, y_{stream}

$local_{u_x}$,$local_{u_y}$,$local_\rho$

$local_f[9]$, f_{eq}

for $i = 1 \rightarrow 9$ **do**

    $x_{stream} = x - c_x[i]$

    $y_{stream} = y - c_y[i]$

    $local_f[i] = f_1[x][y][i]$

end for

for $i = 1 \rightarrow 9$ **do**

    $local_\rho += local_f[i]$

    $local_{u_x} += c_x[i] \times local_f[i]$

    $local_{u_y} += c_y[i] \times local_f[i]$

end for

$local_{u_x} = local_{u_x}/local_\rho$

$local_{u_y} = local_{u_y}/local_\rho$

for $i = 1 \rightarrow 9$ **do**

    $f_{eq} = \varpi[i] \cdot \rho \cdot (1 + 3 \cdot (c_x[i] \cdot local_{u_x} + c_y[i] \cdot local_{u_y}) + (c_x[i] \cdot local_{u_x} + c_y[i] \cdot local_{u_y})^2 - 1.5 \times ((local_{u_x})^2 + (local_{u_y})^2))$

    $f_2[x][y][i] = local_f[i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$

end for

variables (velocities and density). Two routines are necessary to prevent race conditions. **No entiendo la anterior frase. Hay que argumentarlo mejor**. A schematical sketch of the LBM implementation is given in Algorithm 4.

Algorithm 4 LBM *push*.

1: Collide Stream (u_x,u_y,ρ,f_1,f_2,ϖ,c_x,c_y)
2: x, y
3: x_{stream}, y_{stream}
4: f_{eq}
5: **for** $i = 1 \rightarrow 9$ **do**
6: $f_{eq} = \varpi[i] \cdot \rho \cdot (1 + 3 \cdot (c_x[i] \cdot u_x[x][y] + c_y[i] \cdot u_y[x][y]) + (c_x[i] \cdot u_x[x][y] + c_y[i] \cdot u_y[x][y])^2 - 1.5 \times ((u_x[x][y])^2 + (u_y[x][y])^2))$
7: $f_1[x][y][i] = f_1[x][y][i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
8: $x_{stream} = x + c_x[i]$
9: $y_{stream} = y + c_y[i]$
10: $f_2[x_{stream}][y_{stream}][i] = f_1[x][y][i]$
11: **end for**

1: Macroscopic (u_x,u_y,ρ,f_2,c_x,c_y)
2: x, y
3: $local_{u_x}$,$local_{u_y}$,$local_\rho$
4: **for** $i = 1 \rightarrow 9$ **do**
5: $local_\rho += f_2[x][y][i]$
6: $local_{u_x} += c_x[i] \times f_2[x][y][i]$
7: $local_{u_y} += c_y[i] \times f_2[x][y][i]$
8: **end for**
9: $\rho[x][y] = local_\rho$
10: $u_x[x][y] = local_{u_x}/local_\rho$
11: $u_y[x][y] = local_{u_y}/local_\rho$

Two different *push* approaches arise regarding the position of the macroscopic level computing, *macro-collide-stream* and *collide-stream-macro*. Basically, the position of this step can be carried out at the beginning or at the end of the whole process. As we will see later, this fact has important consequences

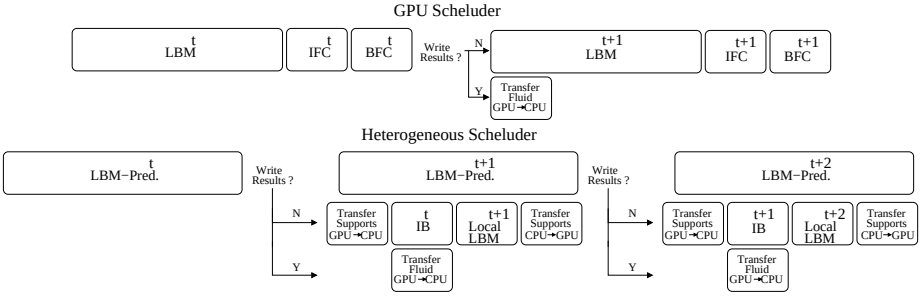

Figure 10: GPU (top) and heterogeneous CPU-GPU (bottom) implementations.

on the performance when implemented on our target architectures.

5. Coupled LBM-IB on Heterogeneous Platforms

After describing different implementation and strategies for LBM and IB as isolated entities, we describe next two different approaches towards the integration of both techniques on heterogeneous architectures.

5.1. Coupled LBM-IB on hybrid CPU-GPU platforms

Our first parallel implementation of the LBM-IB method performs all the major steps on the GPU. The host CPU is used exclusively for a pre-processing stage that sets up the initial configuration and uploads those initial data to the GPU memory and a monitoring stage that downloads the information of each lattice node (i.e., velocity components and density) back to the CPU memory when required. As shown in Figure 10 (top), this implementation consists of three CUDA kernels denoted as LBM, IFC and BFC respectively, that are launched consecutively for every time step. The first kernel implements the LBM method while the other two perform the IB correction. The overhead of the preprocessing stage performed on the CPU is negligible and the data transfer of the monitoring stage are mostly overlapped with the execution of the LBM kernel.

Although this approach achieves satisfactory results, its speedups are substantially lower than those achieved by pure LBM solvers [12]. The obvious

reason behind this behavior is the ratio between the characteristic volume fraction and the fluid field, which is typically very small. Therefore, the amount of data parallelism in the LBM kernel is substantially higher than in the other two kernels. In fact, for the target problems investigated, millions of threads compute the LBM kernel, while the IFC and BFC kernels only need thousands of them. But in addition, those kernels also require *atomic* functions due to the intrinsic characteristics of the IB method, and those operations usually degrade performance.

As an alternative to mitigate those problems, we have explored a heterogeneous implementation graphically illustrated in Figure 10 (bottom). The LBM kernel is computed on the GPU as in the previous approach but the whole IB method and an additional local correction to LBM on the supports of the *Lagrangian* points are performed on the CPU in a coordinated way using a pipeline. This way, we are able to overlap the prediction of the fluid field for the $t+1$ iteration with the correction of the IB method on the previous iteration t at the expense of a local LBM computation of the"$t+1$" iteration on the CPU and additional transfers of the *supports* between the GPU and the CPU at each simulation step.

5.2. Coupled LBM-IB on hybrid CPU-Intel Xeon Phi platforms

We present next an alternative hybrid implementation targeting heterogeneous CPU-Intel Xeon Phi platforms, which favors the particular features of this type of systems. Unlike the previous platform, this approach is more suitable for a proper integration among both architectures, multi-core and Intel Xeon Phi. This approach attempts to take advantages of the transparency that this system offers in terms of programmability, being the communication among both architectures simpler. One of the main claims of Intel with respect to these platforms consists of intending that the sections to be executed on multi-core and the Intel Xeon Phi are as similar as possible. In order to follow these guidelines, a higher importance is given to multicore architecture, as it shows a profitable performance computing LBM, instead of computing only IB.

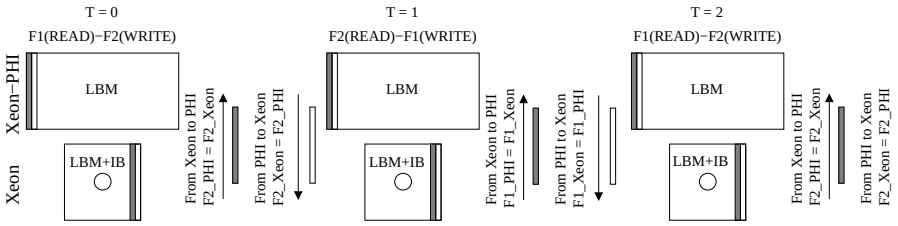

Figure 11: Hybrid approach.

This new strategy basically consists of breaking the data dependency among LBM and IB method by dividing the whole domain into two sub-domains, where the LBM and IB method are solved in one of these domains, while the rest of the fluid domain is solved in the another one. According to the features of both algorithms, LB and IB, and architectures, Intel Xeon and Intel Xeon Phi, we have proposed the approach illustrated in Figure 11.

Next, it is detailed the main features of this approach focusing on the distribution of the computational load, as well as the inter-architectures communication. As known, the performance in term of GFLOPs shown by Phi is higher (around 3× higher) with respect to multicore architectures. Furthermore, the IB method presents some drawbacks to be efficiently managed on massively parallel architectures.

For the purpose of favoring the overlapping space, taking into account the performance of both architectures, the influence of the solid and a piece of fluid domain (around 25% in relation to the entire one domain) is computed on multicore, while the rest of the domain is solved on Phi. Obviously, the hybrid approach requires an additional cost with respect the homogeneous strategies in terms of memory transfers. In particular, the boundary regions among both domains are in need to be communicated each other from (to) both memories spaces. No additional communications are needed since each architecture works on its own space of memory (domain). The communication is based on transferring a common space composed by two columns, one for reading and one for computing. These two columns are shared by both architectures. However, the

column to be computed on multicore (gray column in Figure 11) is read by Phi, and vice-versa (white column). These transfers involve a very small portion of the domain, and so they are very fast. In addition, as depending on the memory layout selected, the elements of the shared columns could not be contiguous in memory, they are packed/unpacked prior to/after each data transfer between memory spaces.

As described previously, we make use of 2 lattices (*f1 and f2*) in order to be able to compute in only single-loop function all the LBM steps, where each time step read from one copy and write results to the other. The communication is carried out at the end of each time step. It consists of overwriting those boundary regions which are in need to be communicated, that is the gray column in the Xeon domain and the white column in the Phi domain. In particular, it is overwritten the lattice updated in the current time step (*f2/f1* for *even/odd* time steps).

6. Performance Evaluation

6.1. Experimental setup

To critically evaluate the performance of the developed LBM-IB solvers, next we consider a number of tests executed on two different heterogeneous architectures, CPU-GPU (i.e., Xeon-Kepler) and CPU-Phi system. More details about the specific architectures that have been used for performance evaluation are given in Table 2. According to the memory requirements of the kernels, the memory hierarchy of the GPU has been configured as 16KB shared memory and 48KB L1, since our codes do not benefit from a higher amount of shared memory on the investigated tests. All the simulations have been performed using double precision and as a performance metric we have used the conventional MFLUPS metric (millions of fluid lattice updates per second) used in most LBM studies.

6.2. Immerse boundary performance evaluation

The first tests (Figure 12) focus exclusively on the IB method using a synthetic simulation without considering the LBM method and analyze its acceler-

Platform	Intel Xeon	NVIDIA GPU	Intel Xeon Phi
Model	E5520/E5-2670	Kepler K20c	5510P
Frequency	2.26/2.6 GHz	XX	1.053 Ghz
Cores	8/16	2496	60
On-chip Mem.	L1 32KB (per core)	SM 16/48KB (per MP)	L1 32KB (per core)
	L2 512KB (unified)	L1 48/16KB (per MP)	L2 256KB (per core)
	L3 20MB (unified)	L2 768KB (unified)	L2 30MB (coherent)
Memory	64/32GB DDR3	5GB GDDR5	8GB GDDR5
Bandwidth	51.2 GB/s	208 GB/s	320 GB/s

Table 2: Summary of the main features of the platforms used in our experimental evaluation. (TODO: check data!!)

ation on both multicore and GPUs. Even for a moderate number of *Lagrangian* nodes, we achieve substantial speedups over the sequential implementation on both platforms. Despite the overheads mentioned above, our GPU implementation is able to outperform the multicore counterpart (8 cores) from 2500 *Lagrangian* markers on.

6.3. Lattice-Boltzmann performance evaluation

The aforementioned alternatives for implementing the LBM are evaluated for the next architectures: Intel Xeon, NVIDIA Kepler and Intel Xeon Phi. In particular, we start implementing several approaches for NVIDIA Kepler architecture, as it is the hardware platform on which most LBM approaches have been implemented. Also, we evaluate a new open-source tool called *sailfish* [34], which includes a LBM code based on the *push*-LBM (macro-collide-stream) scheduler. NVIDIA GPUs require CUDA for exposing and exploiting parallelism for general-purpose computations, and usually requires a deep knowledge of the underlying architecture. On the other hand, the codes implemented on Intel Xeon architecture can be easily adapted on the Intel Xeon Phi to meet the particularities of this many-core architecture. Vectorization is the main recommendation from Intel for boosting the performance. Basically, the main difference from programmer point of view is the size of the vector units, 256-bit

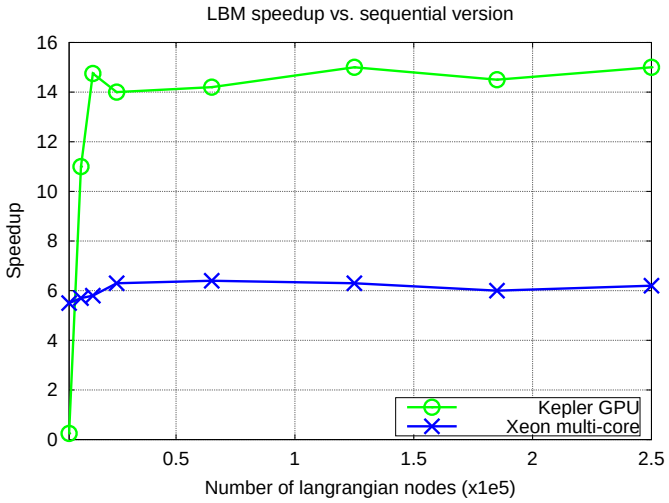

Figure 12: Speedups of the IB method on multicore and GPU for increasing number of Lagrangian nodes.

for Intel Xeon and 512-bit for Intel Xeon Phi. Although many approaches exist to achieve and efficient vectorization (e.g. Cilk Plus, Array Notation, SIMD directives or Auto-vectorization), we have mainly explored Auto-vectorization as a way to maintain a common code baseline among Intel architectures, guiding the compiler towards an efficient SIMD exploitation. In the following, we explore and illustrate some of the tuning approaches to adapt the LBM to our platforms.

6.3.1. LBM on the NVIDIA Kepler GPU

Based on insights extracted from a number of previous works [9, 10, 11, 12, 13], we have focused on analyzing the performance of LBM on GPU by exploiting the coalesced memory mapping presented in the previous section, and a thread mapping consisting of one thread per lattice unit, leaving proved the efficiency of this approach. In the following, we will analyze both LBM approaches: *push* and *pull*. In particular, two alternative *push* approaches, *macro-collide-stream* and *collide-stream-macro*, are evaluated. Although the *pull* approach offers several advantages over the *push* approaches, the latter are

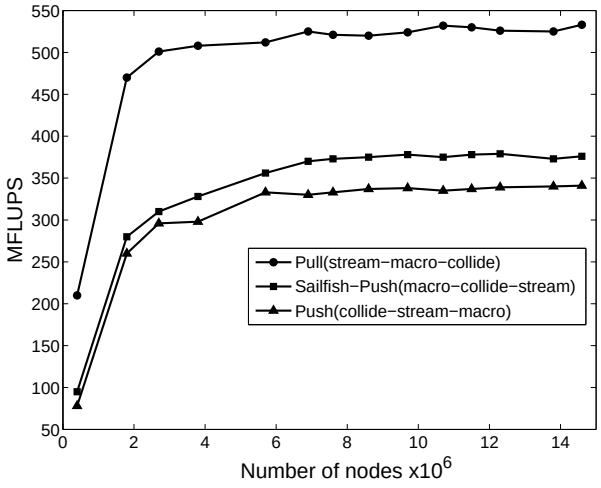

Figure 13: Performance of the *push* (bottom-line) and *pull* (top-lines) LBM implementations on NVIDIA Kepler GPU.

commonly required numerically for some alternative implementations, such as *multi-domain refinement* [33], so they are also worth a special attention.

Obviously, a strong point of synchronism among the collide-stream and macroscopic steps of LBM and a higher number of accesses to global memory degrade the performance of the *push* approaches over the *pull* strategy. More specifically, the *macro-collide-stream push* scheme achieves better performance with respect to the *collide-stream-macro push* one. This is mainly due to the position of the macroscopic level computation. **No entiendo lo anterior!!!...** In both schemes, it becomes mandatory to add a synchornization point among macroscopic and mesoscopic (collide and stream) levels; however, computing the macroscopic level at the beginning allows us to use just one kernel instead of two as in the *collide-stream-macro* scheme. This is translated into a lower number of CPU-GPU communications and reduces the overhead of kernel invocations, and thus the impact on performance. Figure 13 graphically illustrates the performance achieved by both schedulers on our NVIDIA Kepler GPU. The *pull* implementation outperforms the best *push* implementation with a gain around

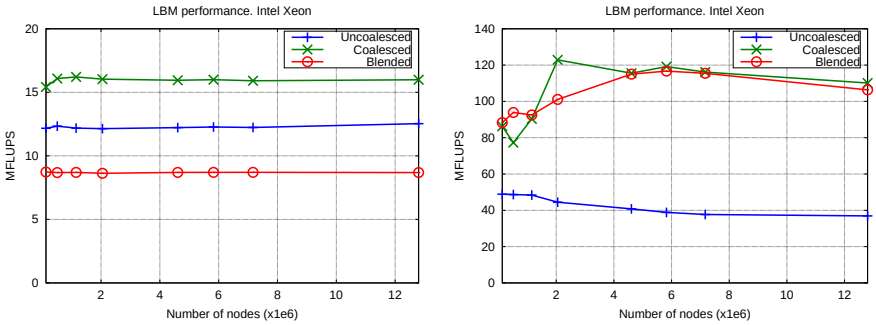

Figure 14: Impact of different memory storage approaches on the Intel Xeon architecture. Left: using 1 thread. Right: using 32 threads.

1.5× in terms of MFLUPS.

6.3.2. LBM on the Intel Xeon

After verifying the superiority of the *pull* LBM approach over its main alternatives in terms of performance, we will also focus on the use of this approach for the rest of the architectures targeted in the paper. **Deberiamos hablar de la estrategia de paralelizacion sobre multi/manycore**. Besides the chosen parallelization scheme, we will mainly focus on the difference between each memory access strategies presented in previous sections. Here, we evaluate the three memory strategies (*uncoalesced*, *coalesced* and *bleded/hybrid*) and their impact on performance. To show the scalability of each approach and the influence on the parallelism, we have run two tests which make use a sequential and a parallelized implementation of the *pull* approach on a dual-socket CPU with 8 cores per chip.

Figure 14 graphically illustrates the performance achieved by each strategy. The *coalesced* approach turns to be the most efficient strategy, using one single hardware thread. In contrast, the *hybrid* and *coalesced* approaches are the best choices when executing multiple threads. Both versions, sequential and parallel, present the lowest performance for the *uncoalesced* memory mapping, which invalidates it as a feasible solution on this type of architectures. Our parallel implementations achieved a speedup around 6× and 8× by implementing the

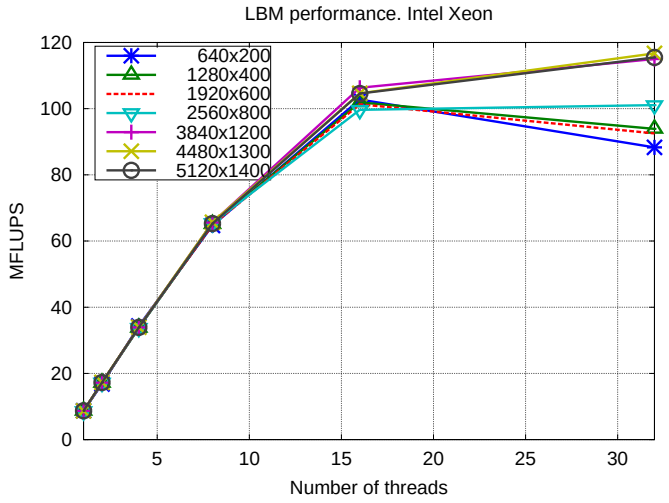

Figure 15: Performance of the LBM implementation using different number of threads on the Intel Xeon.

coalesced and *hybrid* approaches with respect to the sequential counterpart respectively. For both approaches, the memory has been adapted for the size of vector units. **No entiendo lo anterior**

Focusing on scalability, Figure 15 illustrates the trend in performance by increasing the number of threads for the *coalesced* memory mapping. In particular, a constant trend is shown by increasing the number of thread up to achieve 32 threads, with small differences in performance when varying the grid size. However, from 16 hardware threads on, only those problems which present a sufficient workload (that is, grid size) are amenable to continue increasing the performance.

6.3.3. LBM on the Intel Xeon Phi

As for the Intel Xeon, we first evaluate the different memory mappings on Intel Xeon Phi. Figure 16 shows the efficiency obtained by each strategy. Contrary to what has been achieved in multicore CPU, the *hybrid* approach achieves here a slight gain over the *coalesced* one.

One of the common tuning options that Intel Xeon Phi exposes is the selec-

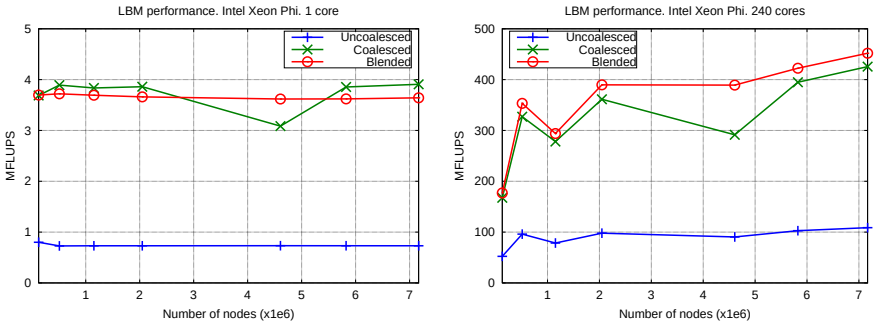

Figure 16: Impact of different memory storage approaches on the Intel Xeon Phi architecture. Left: using 1 thread. Right: using 240 threads.

tion of the thread-core affinity. In our case, three different pre-defined affinity strategies have been studied: *scatter, balanced and compact* (see [**?**] for details on the rationale of each strategy). Taking into account that the best choice for our problem, that is the *hybrid* approach, the three thread-core distributions are evaluated for it. Figure 17 shows the performance achieved by the different strategies, being the most efficient the *compact* one; differences between them (in terms of performance) are reduced and not significant.

After verification of the best strategy for memory mapping and core affinity, we carry out a study about the scalability over the same test bed previously evaluated by the multicore CPU architecture (Figure 18). Except for the smallest grid sizes, a common trend is shown for the most scenarios, a higher number of threads achieves a better performance. In general, comparing the Xeon Phi implementation with its counterpart on the general-purpose multi-core CPU, our approach achieves a gain around 25× and 4× with comparing with the sequential and multi-threaded (32 threads) codes respectively, in terms of MFLUPS.

6.4. Coupled LBM-IB on heterogeneous platforms

After evaluating the performance of both methods (LBM and IB) in an isolated way, we present the strategies carried out for the implementation of the whole fluid-solid solver, adapted to hybrid architectures based on multi-core

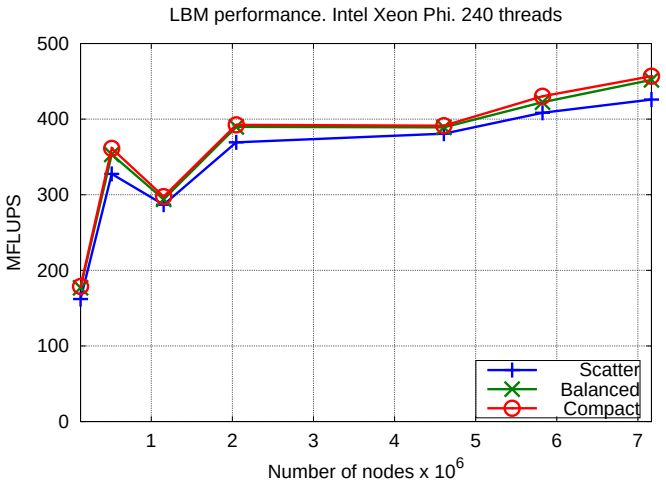

Figure 17: Performance of the LBM implementation using different thread-core affinity strategies on the Intel Xeon Phi.

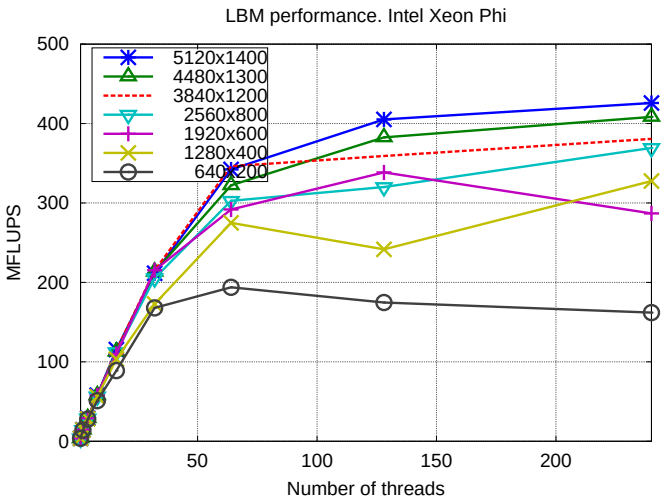

Figure 18: Performance of the LBM implementation using different number of threads on the Intel Xeon Phi.

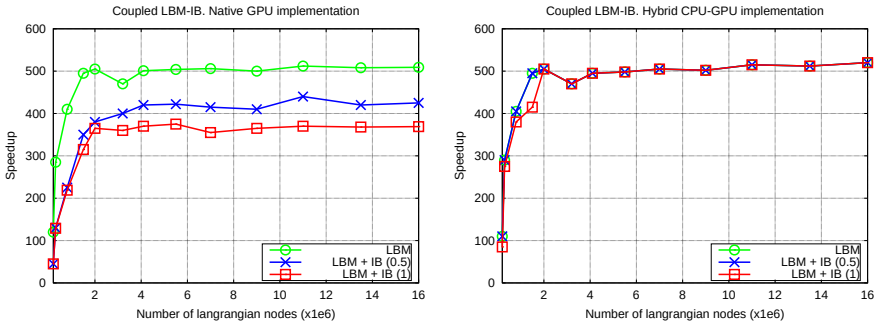

Figure 19: Performance of our GPU (left) and CPU-GPU (right) solvers in MFLUPS for the investigated simulations.

CPUs, the first accelerated by a Nvidia Kepler GPU, and the second by an Intel Xeon Phi.

6.4.1. LBM-IB on hybrid CPU-GPU architectures

The performance of the complete LBM-IB solver is analyzed in Figure 19. We have used the same physical setting as in Section 2 for an increasing number of lattice nodes to analyze the scalability of the method. We have investigated two realistic scenarios with characteristic volume fractions of 0.5% and 1% respectively (i.e. the amount of embedded *Lagrangian* markers also grows with the number of lattice nodes).

The performance of the homogeneous GPU implementation of the LBM-IB method drops substantially over the pure LBM-*pull* implementation (Figure 19). The slowdown is around 15% for a solid volume fraction of 0.5%, growing to 25% for the 1% case. In contrast, for these fractions our heterogeneous approach is able to hide the overheads of the IB method, reaching similar performance to the pure LBM-*pull* implementation.

Figure 20 illustrates the overhead of the IFC and BFC kernel in the GPU homogeneous approach (*left*) and the execution time consumed by the IB method and the local LBM corrections (*right*) over the pure LBM implementation in the heterogeneous approach for a solid volume fraction of 1%. As shown, the consumed time by the steps computed on multi-core (i.e. IB method and local

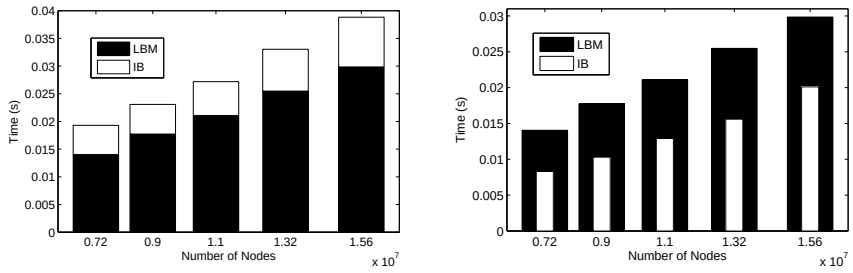

Figure 20: Execution time consumed by the LBM and IB method on both, the GPU homogeneous (left) and multicore-GPU heterogeneous (right) platforms.

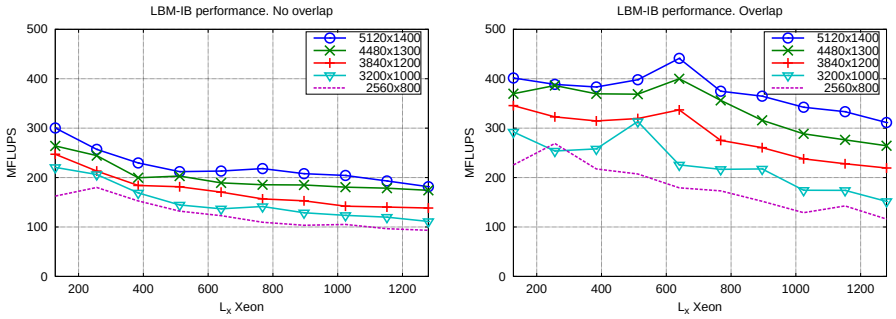

Figure 21: LBM with Immerse Boundary using the Intel Xeon and Intel Xeon Phi, without overlapping (left) and with overlapping (right).

LBM corrections) for the heterogeneous approach does not suppose an additional cost over the pure LBM solver, representing around 65% of the total time consumed by the LBM kernel.

6.4.2. LBM-IB on hybrid CPU-Xeon Phi architectures

Similarly, the performance of the proposed strategy over the CPU-Xeon Phi heterogeneous platform is analyzed. In particular, we have focused on the same numerical scenario with a volume fraction equal to 1%. We exploit the *pull*-coalesced memory mapping on the multi-core and the *pull*-hybrid approach on the Intel Xeon Phi.

In general, a substantial gain in performance is achieved by the heterogeneous-overlapped approach (Figure 21-right) with respect to the heterogeneous-non

overlapped one (Figure 21-left); as an example, for the largest grid tested, the non-overlapped approach attains a peak of roughly 300 MFLUPS, while the counterpart operlapped implementation improves this number up to roughly 450 MFLUPS. As in the CPU-GPU approach, the performance achieved by this CPU-Xeon Phi heterogeneous implementation roughly matches the performance achieved by the pure-LBM implementation (Figure 18), and thus, the impact of the introduction of the solid interaction is mainly hidden.

In this approach, an approapriate work distribution between both platforms is mandatory; in order to find the best load balancing among the portion of the domain executed on the multicore and the portion executed on the Intel Xeon Phi, we have visualized the trend of performance by increasing the size of the domain executed on multi-core (and thus, equivalently, decreasing the amount of work delivered to the Intel Xeon Phi).

If a non-overlapping approach is to be taken (left plot in Figure 21), the weak performance of the multi-core compared with the Intel Xeon Phi, together with the penalty introduced by the PCI-e bus for data transfers between memory spaces, makes it unfeasible to introduce the multi-core as a processor for part of the computation; see, also, that this effect is even more evident as the size of the (sub-)grid processed by the multi-core increases. The conclusions for the overlapped implementation (right plot in Figure 21) are dramatically different. In this case, both platforms can co-operate to solve the problem, and data transfers penalty is hidden if conveniently orchestrated. In fact, the only necessity is for the workload to be correctly balanced. Thus, the percentage of the grid assigned to the multi-core and to the Intel Xeon Phi should be approximately constant, and depends on the grid size. See, for example, how the peak performance for the largest grid tested, where $L_x = 5120, L_y = 1400$ is attained when the portion of the grid assigned to the multi-core is around $L'_x = 620, L_y = 1400$ (that is, around 12% of the complete grid); for the smallest grid tested, where $L'_x = 2560, L_y = 800$, the peak is attained when $L'_x = 250, L_y = 800$, which roughly means a 10% of the complete grid.

7. Conclusions

In this paper, we have investigated the performance of a coupled Lattice-Boltzmann and Immersed Boundary method that simulates the contribution of solid behavior within an incompressible fluid.

We have presented different strategies to enhance LBM performance over three current parallel architectures, Intel Xeon, NVIDIA Kepler GPU and Intel Xeon Phi. Special emphasis has been put on memory management within the many-core platforms, hiding of the impact of data transfers, and on the introduction of different numerical approaches. A more detailed study is carried out on Intel Xeon Phi, as few current works focus on LBM implementations, and even less on coupled LBM/IB approaches. On the other hand, it is possible to find a number of LBM solvers based on NVIDIA GPU architectures. In this sense, our multicore approach achieves a reasonable performance by considering a *coalesced-pull* approach. It obtains a peak performance around 120 MFLUPS by using a dual socket multi-core CPU. Although the Intel Xeon Phi shows a much higher performance by using a *hybrid-pull* scheme with respect to the *coalesced-pull* approach on multicore, being around 3.75× faster, the strategy based on *coalesced-pull* implemented on GPU outperforms these results with a extra benefit around 17%.

In addition, we have focused on the design and analysis of a hybrid implementation that takes advantage of both the accelerator (GPU/Xeon Phi) and multicore in a co-operative way. For realistic physical scenarios with realistic solid volume fractions, our heterogeneous solvers are able to hide the overheads introduced by the IB method and match the performance (in terms of MFLUPS) of state-of-the-art pure LBM solvers.

Our target problem exhibits a dynamic behavior (i.e. its computational cost varies through the time). However, we could take advantage from this characteristic by distributing those stages with a low computational cost (mainly the IB computation) over the general-purpose multi-core, and those with a high computational cost (LBM) over the accelerator, overlapping both computations

in time. In particular, we have proposed two different strategies, one for CPU-GPU architectures and the second for CPU-Xeon Phi, promoting the particular features that each platform presents.

As a future research topic we plan to investigate more complex physical scenarios that require higher amount of memory, making mandatory the use of distributed memory architectures equipped with multi-GPU/Phi platforms.

Acknowledgements

This research was supported by the Basque Government through the BERC 2014-2017 program, by Spanish Ministry of Economy and Competitiveness MINECO: BCAM Severo Ochoa excellence accreditation SEV-2013-0323, by the Spanish governments research contracts TIN2012-32180, CICYTDPI2010-20746 and the Ingenio 2010 Consolider ESP00C-07-20811. We also thanks the support of the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT).

References

[1] C. S. Peskin. The immersed boundary method. Acta Numerica 11, 479-517, 2002.

[2] J. Wu and C.K. Aidun. Simulating 3D deformable particle suspensions using lattice Boltzmann method with discrete external boundary force. Int. J. Numer. Meth. Fluids 62, 765-783, 2010.

[3] W.-X. Huang, S. J. Shin and H J. Sung. Simulation of flexible filaments in a uniform flow by the immersed boundary method. Journal of Computational Physics 226 (2), 2206-2228, 2007.

[4] L. Zhu, C. S. Peskin. Interaction of two flapping filament in a flow soap film. Physics of fluids, 15, 1954-1960, 2000.

[5] L. Zhu, C. S. Peskin. Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method. Physics of fluids, 179, 452-468, 2002.

[6] M. Uhlmann. An immersed boundary method with direct forcing for the simulation of particulate flows. Journal of Computational Physics, 209 (2), 448-476, 2005.

[7] A. Pinelli, I. Naqavi, U. Piomelli, J. Favier. Immersed-Boundary methods for general finite-differences and finite-volume Navier-Stokes solvers. Journal of Computational Physics 229 (24), 9073-9091, 2010.

[8] A. M. Roma and C. S. Peskin and M. J. Berger. An adaptive version of the immersed boundary method. Journal of Computational Physics. 153, 509 - 534, 1999.

[9] J. Tolke, Implementation of a Lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA. Comput. Visual. Sci. 13(1):29-39, 2010.

[10] Y. Zhao, Lattice Boltzmann based PDE Solver on the GPU. Vis. Comput. 24(5):323-333, 2007.

[11] M. Bernaschi, M. Fatica, S. Melchiona, S. Succi, E. Kaxiras. A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. Concurrency Computa.: Pract. Exper. 22, 1-14, 2010.

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

[13] H. Zhou, G. Mo, F. Wu, J. Zhao, M. Rui, K Cen. GPU implementation of lattice Boltzmann method for flows with curved boundaries. Comput. Methods Appl. Mech. Engrg. 225-228, 2012.

[14] S. Xu, Z. J. Wang. An immersed interface method for simulating the interaction of a fluid with movies boundaries. J. Comput. Phys, 216 (2), 454-493, 2006.

[15] D. Calhoun. A Cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. J. Comput. Phys, 176 (2), 231-275, 2002.

[16] D. Russell, Z. J. Wang. A Cartesian grid method for modelling multiple moving objects in 2D incompressible viscous flows. J. Comput. Phys. 191, 177-205, 2003.

[17] A. L. F. L. Silva, A. Silveira-Neto, J. J. R. Damasceno. Numerical simulation of two-dimensional flows over circular cylinder using immersed boundary method. J. Comput. Phys. 189, 351-370, 2003.

[18] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, Manuel Prieto-Matías. Block Tridiagonal Solvers on Heterogeneous Architectures. The 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA). 609-616, 2012.

[19] Pedro Valero-Lara, Alfredo Pinelli, Manuel Prieto-Matias. Fast finite difference Poisson solvers on heterogeneous architectures. Computer Physics Communications. 185(4), 1265-1272, 2014.

[20] Pedro Valero-Lara, Alfredo Pinelli, Manuel Prieto-Matías. Accelerating Solid-fluid Interaction using Lattice-boltzmann and Immersed Boundary Coupled Simulations on Heterogeneous Platforms. Proceedings of the International Conference on Computational Science (ICCS). 50-61, 2014.

[21] Z. Guo, C. Zheng and B. Shi. An extrapolation method for boundary conditions in lattice Boltzmann method. Phys. Fluids, 14 (6), 2007-2010, 2002.

[22] S. Succi. The lattice Boltzmann equation: for fluid dynamics and beyond. Oxford university press New York, 2001.

[23] O. Malaspinas and P. Sagaut. Consistent subgrid scale modelling for lattice Boltzmann methods. Journal of Fluid Mechanics, 700 (1), 514-542, 2012.

[24] S. Marié, D. Ricot and P. Sagaut. Comparison between lattice Boltzmann method and Navier–Stokes high order schemes for computational aeroacoustics. Journal of Computational Physics, 228 (4), 1056-1070, 2009.

[25] P. Bhatnagar, E. Gross and M. Krook. A model for collision processes in gases. I: small amplitude processes in charged and neutral one-component system. Physical Review, 94, 511-525, 1954.

[26] X. He and L. Luo. A priori derivation of the lattice Boltzmann equation. Physical Review, 55 (6), 1997.

[27] Y. Qian, D. D'Humieres and P. Lallemand. Lattice BGK Models for Navier-Stokes Equation. Europhysics Letters, 17 (6), 479-484, 1992.

[28] G. Wellein, T. Zeiser, G. Hager and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. Computers & Fluids, 35, 910-919, 2006.

[29] J. Favier, A. Revell and A. Pinelli. A lattice boltzmann - immersed boundary method to simulate the fluid interaction with moving and slender flexible objects. HAL hal(00822044), 2013.

[30] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, M. Krafczyk. Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for {CPUs} and {GPUs}. Mesoscopic Methods for Engineering and Science - Proceedings of ICMMES-09 Mesoscopic Methods for Engineering and Science, 61 (12), 3730 - 3743, 2011.

[31] S. K. Laytona, Anush Krishnana, L. A. Barbaa. cuIBM - A GPU-accelerated Immersed Boundary Method. 23^{rd} International Conference on Parallel Computational Fluid Dynamics (ParCFD), 2011.

[32] K. Taira, T. Colonius. The immersed boundary method: A projection approach. Journal of Computational Physics. 225 (2), 2118 - 2137, 2007.

[33] D. Lagrava, O. Malaspinas, J. Latt and B. Chopard. Advances in multi-domain lattice Boltzmann grid refinement. Journal of Computational Physics, 231, 4808-4822, 2012.

[34] Sailfish-Team. Lattice Boltzmann (LBM) simulation package for GPUs (CUDA, OpenCL). `https://github.com/sailfish-team/`,2014.