Accelerating Solid-Fluid Interaction using Lattice-Boltzmann and Immersed Boundary Coupled Simulations on Heterogeneous Platforms *

Pedro Valero-Lara[1], Alfredo Pinelli[2], and Manuel Prieto-Matias[3]

[1] Simulation Unit, Research Center for Energy, Environment and Technology, Madrid, Spain.
pedro.valero@ciemat.es
[2] School of Engineering and Mathematical Sciences, City University London, London, United Kingdom.
alfredo.pinelli.1@city.ac.uk
[3] School of Computing, Complutense University of Madrid (UCM), Madrid, Spain.
mpmatias@dacya.ucm.es

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. The proposed method uses a Cartesian uniform grid that incorporates both the fluid and the solid domain. This is a very optimum and novel method to solve this problem and is a growing research topic in Computational Fluid Dynamics. We explain in detail the parallelization of the whole method on both GPUs and an heterogeneous GPU-Multicore platform and describe different optimizations, focusing on memory management and CPU-GPU communication. Our performance evaluation consists of a series of numerical experiments that simulate situations of industrial and research interest. Based on these tests, we have shown that the baseline LBM implementation achieves satisfactory results on GPUs. Unfortunately, when coupling LBM and IB methods on GPUs, the overheads of IB degrade the overall performance. As an alternative we have explored an heterogeneous implementation that is able to hide such overheads and allows us to exploit both Multicore and GPU resources in a cooperative way.

Keywords:

1 Introduction

The main objective of this work consists of minimizing the overhead caused by the simulation of solid-fluid interaction on CPU-GPU heterogeneous platforms. In particular, it is proposed

*This research was funded 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).

an CPU-GPU heterogeneous scheduler which distributes either to GPU or CPU the different parts of the whole solver depending on their parallel features.

The dynamics of a solid in flow field is a research topic currently enjoying growing interest in many scientific communities. It is intrinsically interdisciplinary (structural mechanic, fluid mechanic, applied mathematics, etc) and covers a broad range of applications (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 [**?**, **?**, **?**, **?**]. The use of GPU architectures to compute the fluid field is widely used within Computational Fluid Dynamics community due to the significant performance results achieved [**?**, **?**, **?**]. In contrast, the solid-fluid interaction has only recently gained wider interest.

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 these new parallel platforms [**?**, **?**]. A choice that better meets the GPUs characteristics is based on modeling the fluid flow through the Lattice Boltzmann method (LBM). Several recent works have shown that the combination of GPU-based platforms 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 straightfoward way (see for example [**?**, **?**] and references therein). Nevertheless, no much works has been done to 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 [**?**] 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 [**?**]. Here, we will focus on a different approach based on LBM coupled with an Immersed Boundary method technique able to deal with complex, moving or deformable boundaries [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Special emphasis are given to the algorithmic and implementation techniques adopted to keep the solver highly efficient on CPU-GPU heterogeneous platforms.

This paper is structured as follows. Section **??** briefly 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 **??** the specific potential parallel features of IB method are presented. In Section **??**, we detail the parallel strategies envisaged to optimally enhance the performance of the global LBM-IB algorithm on CPU-GPU heterogeneous platforms. Finally, Section **??** details the performance analysis of the proposed techniques and in Section **??** some conclusions are outlined.

2 Mathematical formulation: Lattice Boltzmann and Immersed Boundary Method

The Lattice Boltzmann method combined with an Immersed Boundary technique is highly attractive when dealing with moving or deformable 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 [**?**] for a complete overview) and now is regarded as a powerful and efficient alternative to classical Navier Stokes solvers. 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 that relies upon an unique relaxation time τ toward the equilibrium distribution $f_i^{(eq)}$:

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

The particles can move only along the links of a regular Lattice defined by the discrete speeds $(e_0=c(0,0);\ e_i=c(\pm1,0),c(0,\pm1),i=1\cdots4;\ e_i=c(\pm1,\pm1),c(\pm1,\pm1),i=5\cdots8$ 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* is used , but all the techniques that will be presented can be extended in a straightforward manner to three dimensional lattices. The equilibrium function $f^{(eq)}\left(\mathbf{x},t\right)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution:

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

In equation **??**, c_s is the speed of sound ($c_s=1/\sqrt{3}$) and the weight coefficients ω_i are $\omega_0=4/9$, $\omega_i=1/9,\ i=1\cdots4$ and $\omega_5=1/36,\ i=5\cdots8$ according to the current normalization. The macroscopic velocity $\mathbf{u}$ in equation **??** 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 **??**, 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 [**?**] 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 **??**, neglecting terms of $\mathrm{O}(\epsilon M^2)$ and using expression **??**, 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 **??** is typically advanced in time in two stages, the collision and the streaming ones.

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:

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

Streaming stage:

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

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 [**?**] 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.

We close this section by briefly explaining 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 [**?**, **?**]. In the present 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 set up 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 **??**) 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 **??**) 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 [**?**] and [**?**] 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:

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

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 **??** 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 **??**. More details about the method and in particular about the determination of the ϵ_n values can be found in [**?**]. 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. As already mentioned, the embedded boundary curve is discretized into a number of markers $\mathbf{X}_I$, $I = 1..N_e$. 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 [**?**]

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

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 **??**. The outlined method has been validated for several test cases including moving rigid immersed objects and flexible ones [**?**].

3 Immersed Boundary on Multicore and GPU Platforms

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 of parallel computing. To compute the IB method and the Body Force Introduction (BFI), 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. In order to facilitate memory bandwidth exploitation and the parallel distribution of the workload, memory structures based on the style of C programming language have been used. Two different memory management approaches are proposed depending on the use of multicore or GPU, since both architecture show different memory features and hierarchy.

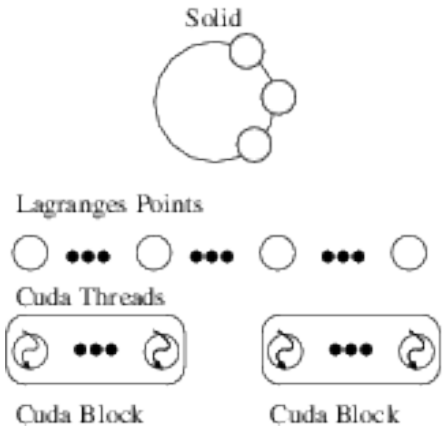

Figure 1: CUDA block-thread distribution for *Lagrangian* points.

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 parallelisms. In contrast, in order to achieve a coalescing access to global memory, the GPU approach distributes the information of all *Lagrangian* points in a set of one-dimensional arrays. In this way, continuous threads access to continuous memory locations.

Next, several approaches to implement the IB method are proposed. 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 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.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 2 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 **??**). 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.The pseudo-code of the IFC kernel is graphically illustrated in Algorithm **??**.

Algorithm 2 BFC kernel.

```
1: BFC_kernel(solid s,f_x,f_y)
2: F_body(Body Force),g (Gravity),x, y,vel_x, vel_y
3: for i = 1 → 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 → 9 do
9:         F_body = (1 − 0.5 · 1/τ) · w[j] · (3 · ((c_x[j] − vel_x) · (f_x[x][y] + g) + (c_y[j] − vel_y) · f_y[x][y]))
10:        AddAtom(f^{n+1}[j][x][y], F_body)
11:    end for
12: end for
```

After the execution of the IB related computations (IFC kernel), the lattice forces as in Equation **??** (Section **??**) 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 **??**.

4 Lattice-Boltzmann & Immersed Boundary on CPU-GPU Heterogeneous Platforms

The actual computational scheduling of LBM is based on the work by [**?**], a novel efficient CUDA implementation based on a *pull* single-loop strategy: each CUDA thread is uniquely dedicated to a single lattice node, 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 global memory to local memory (coalescing accesses) 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 the global memory (coalescing accesses).

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 **??**-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.

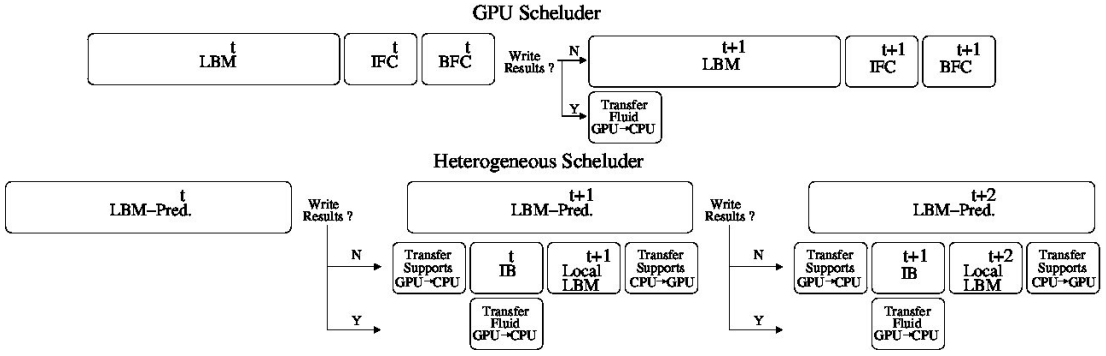

Figure 2: GPU (top) and CPU-GPU Heterogeneous (bottom) implementations.

Although this approach achieves satisfactory results, its speedups are substantially lower than those achieved by pure LBM solvers [**?**]. The obvious reason behind this behavior is the ratio between the characteristic volume fraction and the fluid field, which is 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 degrade performance.

As an alternative to mitigate those problems, we have explored a heterogeneous implementation graphically illustrated in Figure **??**-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 is 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 every simulation step.

5 Performance Evaluation

To critically evaluate the performance of the developed LBM and IB solver, next we consider a number of tests executed on a CPU-GPU (i.e., Xeon-Kepler) system. More details about the specific architectures that have been used for performance evaluation are given in Table **??**. According to the memory requirements of the kernels, the memory hierarchy 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.

Platform	Xeon E5520 (2.26 GHz)	Kepler K20c
Cores	8	2496
on-chip Memory	L1 32KB (per core)	SM 16/48KB (per MP)
	L2 512KB (unified)	L1 48/16KB (per MP)
	L3 20MB (unified)	L2 768KB (unified)
Memory	64GB DDR3	5GB GDDR5
Bandwidth	51.2 GB/s	208 GB/s
Compiler	gcc 4.6.2	nvcc 5.5

Table 1: Details of the experimental platforms.

The first tests (Figure **??**) focus exclusively on the IB method using a synthetic simulation without considering the LBM method and analyze its acceleration 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.

The performance of the whole LBM-IB solver is analyzed in Figure **??**. We have used the same physical setting as in Section **??** with an increasing number of lattice nodes to analyzed the scalability of the method. We have investigated two realistic scenarios with characteristics 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 implementation. The slowdown is around 15% for a solid

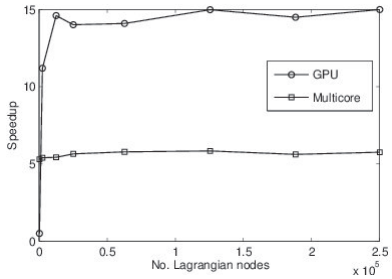

Figure 3: Speedups of the IB method on multicore and GPU for increasing number of Lagrangian nodes.

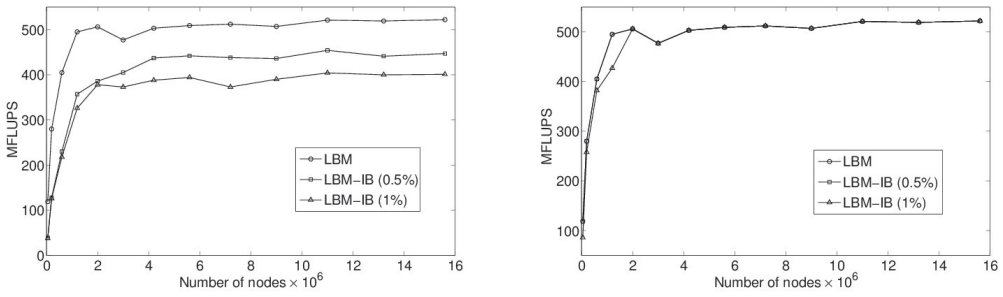

Figure 4: Performance of our GPU (left) and CPU-GPU (right) solvers in MFLUPS for the investigated simulations.

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 implementation.

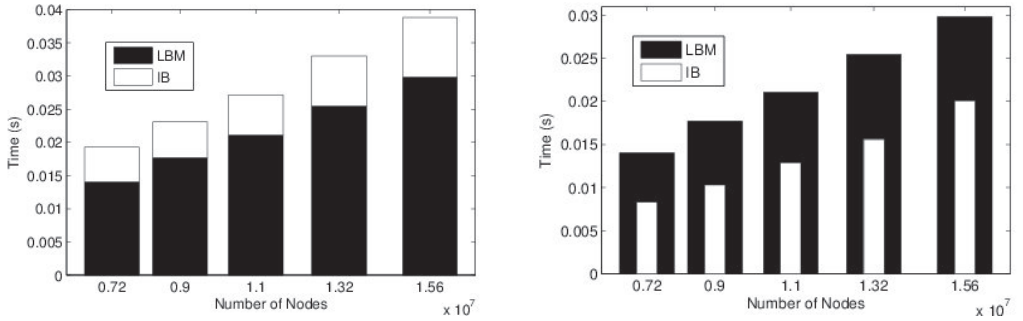

Figure 5: Execution time consumed by the LBM and IB method on both, the GPU homogeneous (left) and multicore-GPU heterogeneous (right) platforms.

Finally, Figure **??** 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 multicore (i.e. IB method and local LBM corrections) in the multicore-GPU 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 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. While, the Lattice-Boltzmann method has been widely studied on heterogeneous platforms, the Immersed-Boundary method has recieved less attention.

Our main contribution is the design an analysis of a heterogeneous implementation that takes advantage of both GPUs and multicore in a cooperative way. For realistic physical scenarios with realistic solid volume fractions, our heterogeneous solver is able to hide the overheads introduced by the Immersed-Boundary method and match the performance (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 were able to take advantage of this feature by mapping those portions of the problem which present a low computational cost (IB) on multicore and those with a high computational cost (LBM) on GPU, and executing them at very same time.

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 multi-GPU platforms. In addition, we plan to analyzr other parallel platforms such as the Intel Xeon Phi, as well as to implement more elaborated strategies for memory management and distribution.

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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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), 2012.

[17] Pedro Valero-Lara, Alfredo Pinelli, Manuel Prieto-Matias. Fast finite difference Poisson solvers on heterogeneous architectures. Computer Physics Communications, 2014. Available online `http://www.sciencedirect.com/science/article/pii/S0010465513004384`.

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

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

[20] 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.

[21] 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.