FULL GPU Implementation of Lattice-Boltzmann Methods with Immersed Boundary Conditions for Fast Fluid Simulations

G Boroni[1,2]*, J Dottori[1,2], P Rinaldi[2,3]
1. CONICET
2. PLADEMA UNCPBA
3. CICPBA

ABSTRACT
Lattice Boltzmann Method (LBM) has shown great potential in fluid simulations, but performance issues and difficulties to manage complex boundary conditions have hindered a wider application. The upcoming of Graphic Processing Units (GPU) Computing offered a possible solution for the performance issue, and methods like the Immersed Boundary (IB) algorithm proved to be a flexible solution to boundaries. Unfortunately, the implicit IB algorithm makes the LBM implementation in GPU a non-trivial task. This work presents a fully parallel GPU implementation of LBM in combination with IB. The fluid-boundary interaction is implemented via GPU kernels, using execution configurations and data structures specifically designed to accelerate each code execution. Simulations were validated against experimental and analytical data showing good agreement and improving the computational time. Substantial reductions of calculation rates were achieved, lowering down the required time to execute the same model in a CPU to about two magnitude orders.

1. INTRODUCTION

Fluid simulation is a computation-intensive task that can generally consume huge amounts of time. However, recent hardware architectures can lead to substantial increases in performance through many-core processors if appropriate schemes are applied. Many fluid simulation models, like Lattice Boltzmann Methods, can be greatly accelerated by the use of Graphic Processors Units (GPU).

These processors are optimized to execute single-instruction multiple-data operations, namely a simple kernel over each element of a large set simultaneously, operating as a co-processor of the host CPU [1]. Initially, developers could take advantage of the GPU power through the graphics pipeline by means of shading languages [2]. Currently, there are parallel computing architectures for GPU programming using high-level languages [3]. For example, NVIDIA and the Portland Group (PGI) have worked in cooperation to develop CUDA FORTRAN language [4].

In some cases the migration to GPU is a simple code translation, but in most cases a new algorithm must be implemented. In particular, a simple Lattice Boltzmann Method (LBM) fluid solver can be adapted to a GPU by translating some operations and replacing the main

*Corresponding Author: gboroni@gmail.com loops by multiple threads. Several researchers have shown that the combination of GPU and parallel LBM is an excellent tool for fast fluid simulations [5][6][7]. However, practical LBM applications require flexible management of boundary conditions and this cannot be achieved with basic LBM solver.

A more evolved method is needed to overcome boundary issues when the geometry of the problem turns complex, like the Immersed Boundary Method (IB).

An efficient iteration procedure for combining LBM with the Immersed Boundary method was presented showing good results for fluid-solid interactions while keeping a flexible implementation [8]. Recent publications show promising results for the IB-LBM coupling on GPU but, in most implementations, only LBM is run on GPU hardware leaving the IB part for the CPU. Other solutions [9] use a single-step explicit IB algorithm [10] to couple with LBM on GPU code.

In the present work, the algorithm combination was refined and implemented in GPU, which considerably reduce simulation time.

2. MATERIALS AND METHODS

2.1. Lattice Boltzmann Method

LBM is basically a mesoscopic kinetic model with a discrete internal velocity variable, whose average magnitudes obey some macroscopic field equations [11]. The method represents the fluid by a set of particle populations that move between cells over a regular grid. Fluid behavior is achieved by operations between these populations on each cell locally. The most common model for 2D simulations is D2Q9 [12] that uses a square lattice with 9 velocity directions (Fig. 1). In the present work, this model is used for solving 2D Navier-Stokes equations [13].

The population particles in node x at time t having velocity e_α, denoted by $f_\alpha(x,t)$ follow the evolution functions

$$f'_\alpha(x,t) = f_\alpha(x,t) + \frac{1}{\tau}\left[f_\alpha^{(0)}(x,t) - f_\alpha(x,t)\right] \tag{1}$$

$$f_\alpha\left(x + e_\alpha \delta t, t + \delta_t\right) = f'_\alpha(x,t) \tag{2}$$

where the equilibrium function $f_\alpha^{(0)}(x,t)$ determines the macroscopic equations that the automata simulates. Equations (1) and (2) are called the collision and streaming step respectively. In the called D2Q9 grid model the index α spans over nine discrete directions (see Fig. 1).

The equilibrium function to simulate the Navier-Stokes through the Bhatnagar–Gross–Krook (BGK) collision operator can be expressed as [11]

$$f_\alpha^{(0)} = w_\alpha \rho\left[1 + 3\left(e_\alpha \cdot u\right) + \frac{9}{2}\left(e_\alpha \cdot u\right)^2 - \frac{3}{2} u \cdot u\right] \tag{3}$$

where

$$w_0 = \frac{4}{9}; w_\alpha = \frac{1}{9}, \quad \alpha = 1,3,5,7; w_\alpha = \frac{1}{36}, \quad \alpha = 2,4,6,8 \tag{4}$$

and

$$\rho = \sum_\alpha f_\alpha, \ \rho u = \sum_\alpha f_\alpha e_\alpha \tag{5}$$

are the macroscopic density and mass flux respectively. The fluid viscosity can be controlled via the relaxation parameter τ, as

$$\nu = \left[\frac{(2\tau - 1)}{6}\right] e^2 \delta_t \tag{6}$$

where e is the lattice velocity, $\delta x/\delta t$.

For problems with external forces, the following term is added to the right side of (2) [14], as

$$f_\alpha\left(x_{ij} + e_\alpha \delta_t, t + \delta_t\right) = f'_\alpha\left(x_{ij}, t\right) + \delta_t g_\alpha\left(x_{ij}, t\right) \tag{7}$$

$$g_\alpha\left(x_{ij}, t\right) = w_\alpha \left\{3 f_{bij} \cdot \left[\left(e_\alpha - u_{ij}\right) + 3\left(e_\alpha \cdot u_{ij}\right) e_\alpha\right]\right\} \tag{8}$$

where f_{bij} is the force applied to each *ij*-cell. The external forcing term g_α given by (7) and (8) has first order convergence [15], which limits the solution to problems with slow moving boundary or flexible boundary with small pressure gradient. For fast moving boundaries or flexible boundaries with large pressure gradients, a higher order method is needed. Chen and Doolen [11] proposed an alternative implicit second-order convergence scheme, replacing (7) by

$$\begin{aligned} f_\alpha\left(x_{ij} + e_\alpha \delta_t, t + \delta_t\right) = & \ f'_\alpha\left(x_{ij}, t\right) \\ & + \frac{\delta_t}{2}\left[g_\alpha\left(x_{ij}, t\right) + g_\alpha\left(x_{ij} + e_\alpha \delta_t, t + \delta_t\right)\right] \end{aligned} \tag{9}$$

Owing to the use of this equation, the final numeric scheme becomes implicit.

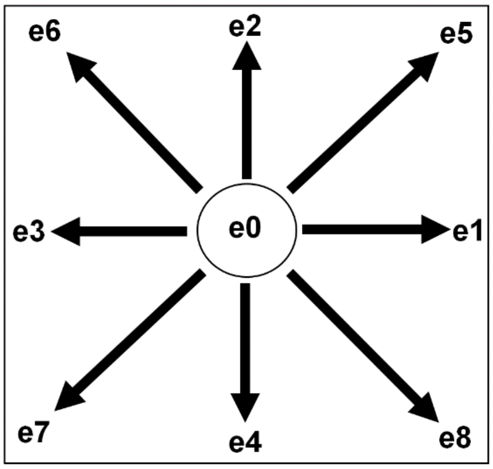

Figure 1: Space of discrete velocities in LBM- d2Q9, corresponding to the population functions f_α.

2.2. The Immersed Boundary Method

The method of Immersed Boundary (IB) was initially developed to deal with flexible boundaries in finite elements method. The boundary is represented by a set of massless particles coupled by elastic forces to space points and between themselves, which moves with the surrounding fluid. Conversely, the force generated by distortions of the boundary is transferred to the fluid [16].

Figure 2 shows a 2D example with a closed immersed boundary. The boundary and the fluid domain are denoted by Γ_b and Ω_f, respectively. The state of the boundary is represented by *X(s,t)*, a Lagrangian vector function of arc length *s* and time *t*, which returns the location of the boundary nodes on Γb. The influence action on the fluid is represented by a force density *F(s,t)* at the boundary point *X(s,t)*. Thus, *F(s,t)* is determined by the configuration of *X(s,t)* and it is transferred into the force term *g* in (8), which determines the flow velocity and pressure throughout domain Ω_f.

For a boundary immersed in a viscous fluid the IB is given by the following set of equations [17]

$$\nabla \cdot u = 0 \tag{10}$$

$$\rho\left(u_t + \left(u \cdot \nabla\right) u\right) = -\nabla p + \mu \cdot \Delta u + f_b \tag{11}$$

$$\begin{aligned} \frac{dX(s,t)}{dt} &= U\left(X(s,t),t\right) \\ &= \int_{\Omega_f} u(x,t)\,\delta\left(x - X(s,t)\right) dx \end{aligned} \tag{12}$$

$$F(s,t) = S_f X(s,t) \tag{13}$$

$$f_b(x,t) = \int_{\Gamma_b} F(s,t)\,\delta(x - X(s,t))\,ds \tag{14}$$

where *u* is the flow velocity, *ρ* the fluid density, *p* the pressure, *υ* the fluid viscosity, f_b the external force, *X* the boundary coordinate, *s* the boundary fiber length, *U* the boundary speed, *x* the fluid flow coordinate, S_f the boundary force generation operator, and *δ(r)* the Dirac delta function. Equations (10) and (11) are the Navier–Stokes equations with external force f_b, while (12) and (13) are the IB equations. Equation (14) and the right part of (12) represent the fluid-boundary interaction. The discretized version of (12) and (14) using a regularized discrete delta function δ_h are

$$f_{bij} = \sum_k F_k \delta_h\left(x_{ij} - X_k\right)\Delta s_k \tag{15}$$

and

$$\frac{dX_k}{dt} = U_k = \sum_{ij} u_{ij}\delta_h\left(x_{ij} - X_k\right)\Delta x \Delta y \tag{16}$$

where *h=Δx=Δy* is the fluid node spacing and *Δsk* is the boundary segment length. The delta function *δh* is an influence distribution given by [17]

$$\delta_h(x,y) = 1/h^2\phi(x/h)\phi(y/h) \tag{17}$$

$$\phi(r) = \begin{cases} (1+\cos(\pi|r|/2))/4 & |r| \le 2 \\ 0 & |r| > 2 \end{cases} \tag{18}$$

The force density *F* induced by the boundary over the fluid is determined by the position of the nodes, and can be written in general as

$$F = k_c \frac{\partial^2 X}{\partial s^2} - k_\gamma \frac{\partial^4 X}{\partial s^4} - k_f\left(X - Z\right) \tag{19}$$

where k_c is the tension stiffness, k_γ the bending rigidity, k_f the fastening stiffness and *Z* the target position of the boundary. The discretized equations of (16) can be expressed as

$$\begin{aligned} F_k = \; & -k_f\left(X - Z\right) \\ & + k_c\left(\frac{X_{k-1} - 2X_k + X_{k+1}}{\Delta s^2}\right) \\ & - k_\gamma\left(\frac{X_{k-2} - 4X_{k-1} + 6X_k - 4X_{k+1} + X_{k+2}}{\Delta s^4}\right) \end{aligned} \tag{20}$$

Two values of F_k corresponding to time in *t-1* and *t* are needed in every step to calculate F_k in an implicit way.

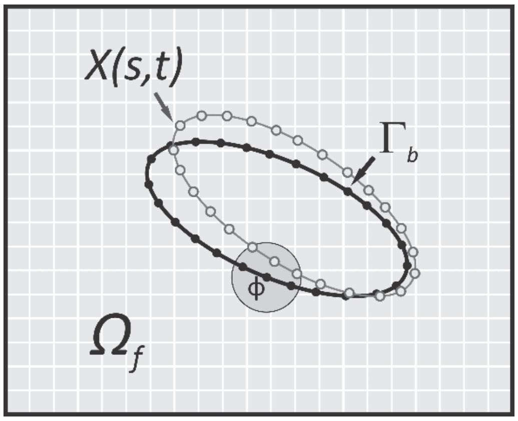

Figure 2: Closed immerse boundary in a 2D lattice.

2.3. GPU implementation of LBM-IB

Since the CPU implementation of LBM with IB was done using FORTRAN programing language, the PGI CUDA FORTRAN compiler [4] was used for the full GPU implementation.

2.3.1. GPU implementation of LBM-IB

The main program is a CPU code that calls the GPU subroutines, called kernels. The basic LBM calculations are applied to each cell of the grid whereas the IB calculations are applied to each point in the boundary [8]. Many authors have suggested that the way to achieve best performances in GPU is by means of a single collide-stream loop [7], [18], [19], but this was studied in the case of LBM with standard boundary conditions. However, when LBM is combined with the IB method, there is a different situation. In effect, the IB coupling introduces an internal iteration that can be combined with the streaming step, but the collision step can be computed only once per time step [8]. Following these execution structure, the proposed implementation of LBM-IB in CUDA pseudo--code is as follows:

```
Algorithm 1 Main Loop
1:  Allocate memory
2:  Initialize equilibrium system variables
3:  Initialize executions configuration variables.
4:  for each time step
5:          Collide  //typical LBM collision
6:          convergence=1
7:          while(convergence==0) //IB loop
8:                  Compute_IB_Points
9:                  Compute_LBM_Forces
10:                 Stream_And_Force
11:                 Compute_Boundaries_And_ Macroscopics
```

where Collide, Compute_IB_Points, Compute_LBM_Forces, Stream_And_Force and Compute_Boundaries_And_Macroscopics are implemented as internal CUDA kernel functions. The execution configuration for each kernel is detailed in the respectively following sections. A global synchronization of threads is performed after each kernel, maximizing parallelism in each computation by making wait the GPU for previous steps data.

2.3.2. Collide kernel

The collision kernel calculates the operator of the LBM scheme given by (1). The execution configuration used for the simulations is computed to maximize parallelism with a single thread per LBM cell.

2.3.3. Compute_IB_Points kernel

This kernel calculates the effect of the fluid cells on the IB. Each boundary point has a 5×5 matrix to store the weight of the force on the neighbor cells (18). The kernel uses a 3D matrix to store the boundary weight values for each boundary point. The matrix is called *BW(dx,dy,k)*, where *k* is the boundary IB-node index and *(dx, dy)* is the cell position respect to the node. Each 2D sub-matrix has five cells in each direction, that is: $[x_{n-2},y_{n-2}\ ;\ x_{n+2},y_{n+2}]$ for boundary point contained in the $[X_n,\ Y_n]$ cell.

There is one GPU thread for each IB-point, which calculates the force exerted to the fluid by the node (15) and the new position of the node (16). The following pseudocode shows the kernel scheme.

```
Kernel 1: Compute_IB_Points
1:  //In each thread
2:  Compute corresponding IB point
3:  Compute new values of equation (20)
4:  For each cell in near region   //Define the lattice region near k
5:              Compute BW Matrix
6:              Compute new position (16)
7:              if (Fk_ant-_Fk(k))>EPSILON) //compare guess of Fk (20)
8:                          convergence=0; //convergence not reached
```

2.3.4. Compute_LBM_Forces kernel

This kernel calculates the force exerted by each IB point on the fluid. It operates as a parallel thread for each LBM cell adding the forces exerted by near IB-nodes (20) weighted with the *BW* matrix. Finally, the kernel calculates the force *g* according to (8). The procedure is as follows

```
Kernel 2: Compute_LBM_Forces
1:  //In each thread
2:  Compute corresponding LBM cell (15)
3:  fbij = 0 //thread local variable
```

```
4: For each IB point
5:         if affects the cell
6:                 fbij += Fk(k)*BW(k,floor(dx),floor(dy));
7: Compute equation (9) based on fbij
```

2.3.5. Stream_And_Force kernel

This function calculates the streaming step of the LBM scheme adding the force applied by the boundary nodes (9). The execution is performed with a single thread per cell.

2.3.6. Compute_Boundaries_And_Macroscopics kernel

This solution kernel runs one thread per cell. Each thread applies standard bounce-back boundary conditions [11] where necessary and calculates the macroscopic variables of the fluid, ρ and u.

2.4. Remarks

The main difference with other IB implementations designed for CPU like the presented in [8] is the convergence condition in the inner loop. Testing the mean error would imply an extra reduction [20], so the maximum error metric is used as a convergence condition. Also the *BW(dx, dy, k)* structure saves the time of computing the $\delta(x,y)$ of each IB point with its surrounding neighborhood each iteration multiple times. In recent GPU implementations, the IB part of the algorithm is processed in CPU [21] or a simplified explicit version is executed in GPU [22].

2.5. Memory usage

Because of the implicit definition of IB algorithms, it is necessary to have memory allocation for both actual time step and next time step. Variables must be minimized in GPU to have double memory space because of limited memory. In this case, the minimum variables are u, LBM forces and also IB Forces to check convergence. Other variables can be avoided to duplicate their memory space.

3. RESULTS

The described IB-LBM algorithm was tested in a GPU NVIDIA GeForce GTX 580 with 3GB DDR5 SDRAM hosted by an AMD PHENOM II X6 2.81GHZ CPU. Two flow cases were simulated to test the performance and accuracy of the scheme, namely, a Poiseuille flow and the vortex shedding behind a solid obstacle.

3.1. Poiseuille flow

The first case is a standard Poiseuille flow between two parallel plates subjected to constant pressure conditions at the inlet and the outlet. In fully developed flow, the velocity profile is parabolic and the shear stress acting on the fluid is exactly balanced by the pressure gradient (Fig. 3), which leads to an analytical solution (11).

Two straight immersed boundaries of *100* points each are located in a 110×61 lattice to represent the channel walls as shown in Fig. 4. The IB lines are separated *5.5* cells away from the grid limits, so they delimit a *50*-cells wide channel between them.

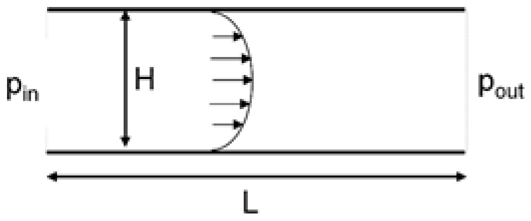

Figure 3: Poiseuille flow in a rectangular channel.

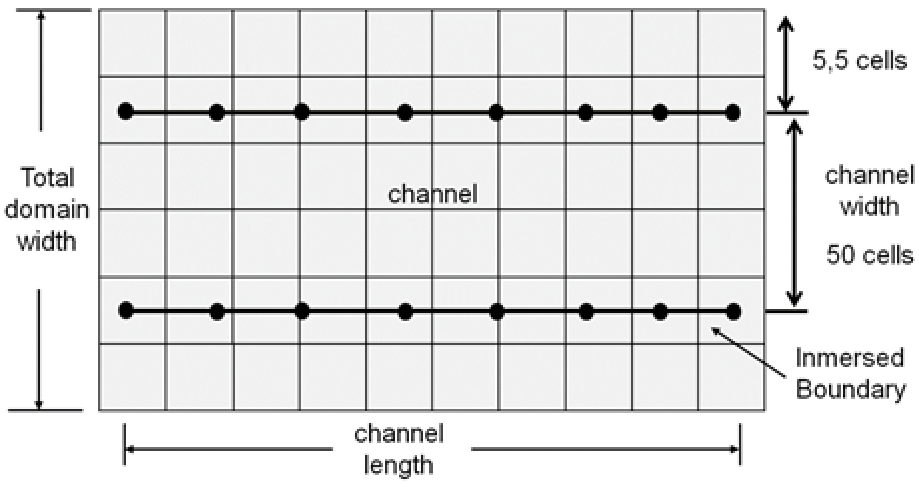

Figure 4: Flow channel represented with IB.

The initial particle density ρ_0 is *1.0* and the density drop between the inlet and the exit of the channel $\Delta\rho$ is 10^{-5} corresponding to $\Delta\rho/\Delta x = 1/3\ 10^{-7}$. The relaxation parameter τ is set in *1.5*, corresponding to a cinematic viscosity $v = 1/3$ in units of $\Delta x^2/\Delta t$, giving a Reynolds number of *46.87*. Fig. 5 shows the excellent agreement between the calculated and analytical velocity profile, with a total quadratic error of $3.3\ 10^{-8}$.

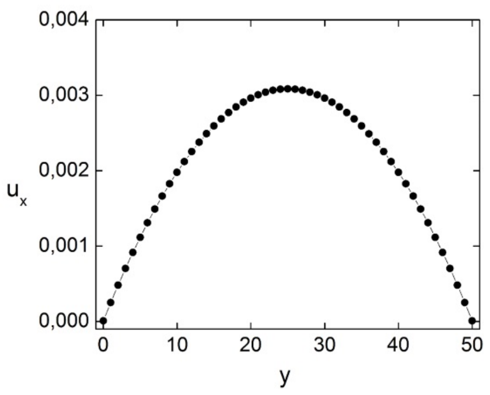

Figure 5: Velocity profile of a Poiseuille, analytical (points) and numerical (curve).

3.2. Vortex shedding

The second test case corresponds to a cylinder located in the center of a channel. At the back of the obstacle, the fluid stream fails to stick to cylinder's wall, and the boundary layers separate from each side of the cylinder resulting in a Von Karman vortex train. A detailed explanation of this case can be found in [8][23].

The channel was represented by a *900×128* lattice, as shown in Fig. 6. The obstacle is a circle, *22*-cells diameter, centered at position *(64, 64)* inside the channel. It is represented by an IB of *200* points, which ensures the no-leakage between points criterion [16]. The channel pressure drop is $\Delta p = 5\ 10^{-4}$, the average particle density is $\rho_0 = 0.05$ and $\tau = 0.65$. Fig. 7 shows the field of the velocity module at $t=5000\Delta t$ steps. The wake of vortexes can be easily recognized. The Strouhal number is $St=fd/V$ where f is the vortex shedding frequency and V the fluid velocity. The obtained St is *0.165*, having a good agreement with available literature [23].

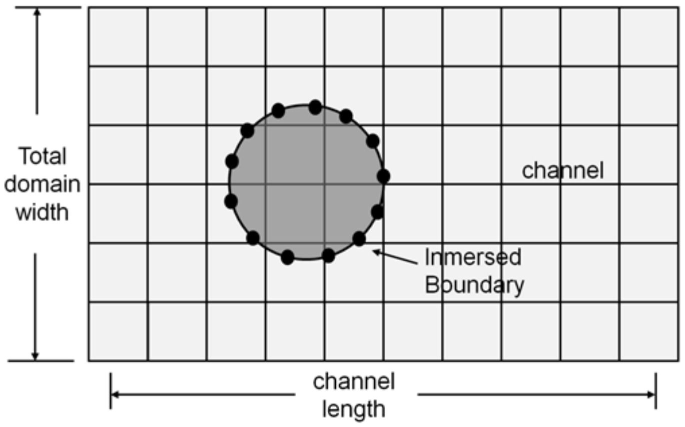

Figure 6: Von Karman vortex street scheme represented with IB.

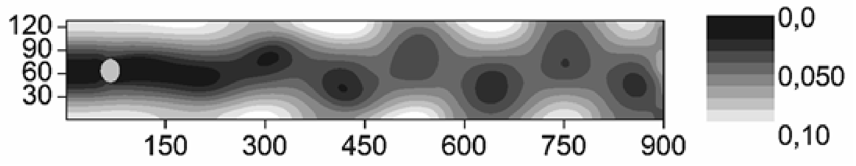

Figure 7: Von Karman vortex street, contour map of velocity module at time step 5000.

3.3. Performance

In order to assess the performance of the present implementation, a comparison was made against an equivalent FORTRAN implementation running in a single thread on a CPU AMD Phenom II X6 at 2.81 GHz. The main algorithm and subroutines are similar to those in GPU code. The main difference is that in the CPU all the work is performed in a single thread fashion. So the subroutines are built as two nested loops for each spatial direction and one loop for the boundary points. Both implementations use single precision floating point representations.

The most popular metric to assess the performance of LBM codes is the number of lattice updates per second (LUPS) [19]. Tables 1 and 2 compare the performances of each implementation for different platforms, scaling the problem to various domain grid sizes. The simulation corresponds to Poiseuille flow case presented in *Section 3.1* (Table 1) and Vortex shedding case presented in *Section 3.2* (Table 2). The number of boundary points used in each case to simulate the channel wall is proportional to the grid length to ensure the no-leakeage between points criterion [16] and it is shown in column 2 of Table 1. It can be seen that the performance achieved with GPU is between *2* and *140* times faster than with CPU. It can be seen that when domain size is increased, the global speed up does so too. In part because the numbers of IB points per cell turns lower, lowering down the computational cost proportion of IB in the whole process.

Table 1: Performance comparison between GPU and CPU codes simulating Poiseuille flow, obtained over 1000 iterations.

Domain Size	**Boundary Points**	**CPU MLUPS**	**GPU CUDA MLUPS**	**Speedup**
50×30 (1500)	80	1.21	2.71	2.23
100×60 (6000)	180	0.95	11.17	11.75
200×120 (24000)	380	0.65	20.68	31.81
400×240 (96000)	780	0.4	29.26	72.13
800×480 (384000)	1580	0.24	20.65	84.28

Table 2: Performance comparison between GPU and CPU codes simulating Vortex shedding, obtained over 1000 iterations.

Domain Size	**Boundary Points**	**CPU MLUPS**	**GPU CUDA MLUPS**	**Speedup**
521×74 (38554)	116	0.95	52.31	55.33
625×89 (55625)	139	0.90	61.64	68.85
750×107 (80250)	167	0.76	69.36	91.28
900x128 (115200)	200	0.65	74.09	113.76
1080×154 (166320)	240	0.57	80.26	140.72

The time consumption of each kernel was also assessed to identify the critical calculation steps. It should be noted that these are just estimations, since the kernels are linked to each other. Table 3 compares the resulting performances of each implementation discriminated by kernels. The Compute_LBM_Forces kernel consumes *90%* of the computing time, being this the critical section of the process. Further optimization strategies can be applied, but probably they will entail resigning some flexibility. For example, sequential kernels with the same execution configuration could be combined like Stream_and_force and Compute_Boundaries_And_Macroscopics but loosing decoupling of code.

Table 3: GPU individual kernel times obtained over 1.000 iterations simulating Poiseuille flow in a grid of 400 x 240 cells.

Kernel	**Execution Time (s)**	**Average Iterations**	**Time per iteration (ms.)**	**Percent**
Collide	0.098	1	0.098	3%
Compute_IB_Points	0.098	2	0.049	3%
Compute_LBM_Forces	2.952	2	1.476	90%
Stream_And_Force	0.033	2	0.016	1%
Compute_Boundaries_And_ Macroscopics	0.098	2	0.049	3%
Total	3.279	1	3.279	100%

4. CONCLUSIONS

A GPU implementation of the Lattice Boltzmann Method for fluid flow combined with immersed boundaries capable of modelling complex and flexible boundary conditions was presented. The IB method represents objects as force fields acting on local neighborhoods around boundaries. On each time step of LBM, the algorithm executes an inner iteration to solve the fluid-boundary interaction implicitly. Data locality and execution scheme are different when standard boundary conditions are used. These are crucial aspects to GPU implementations, so the combination of IB and LBM becomes less trivial to parallelize.

Two classical test case simulations were performed, Poiseuille flow and vortex shedding behind a cylindrical obstacle. The algorithm on GPU can give satisfactory results in terms of accuracy. Substantial reductions of the calculation rates were achieved, reducing up to *84* times the time required by a CPU to execute the same case. The GPU code reaches near *80* Mlups on Geforce GTX 580 desktop graphic board.

REFERENCES

[1] NVIDIA. NVIDIA CUDA C Programing Guide Version 4.0: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. 2010

[2] Mark WR, Glanville RS, Akeley K, Kilgard MJ. Cg: a system for programming graphics hardware in a c-like language. ACM Trans. Graph. 2003; 22(3): 896–907.

[3] NVIDIA. NVIDIA CUDA Home Page : http://developer.nvidia.com/category/zone/cuda-zone; 2007.

[4] PGI. PGI CUDA Fortran Compiler, The Portland Group: http://www.pgroup.com/resources/cudafortran.htm; 2011.

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

[6] Tölke J. Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA. Computing and Visualization in Science. 2010; 13(1): 29–39.

[7] Rinaldi PR, Dari EA, Vénere MJ, Clausse A. A Lattice-Boltzmann solver for 3D fluid simulation on GPU. Simulation Modelling Practice and Theory. 2012; 25: 163-171.

[8] Boroni G, Dottori J, Dalponte D, Rinaldi P, Clausse A. An improved Immersed-Boundary algorithm for fluid-solid interaction in Lattice-Boltzmann simulations. Latin American Applied Research, accepted 2013.

[9] Valero-Lara P, Pinelli A, Prieto-Matias Manuel. Accelerating Solid-Fluid Interaction using Lattice-Boltzmann and Immersed Boundary Coupled Simulations on Heterogeneous Platforms. Procedia Computer Science. 2014; 29: 50-61

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

[11] Chen S, Doolen GD (1998). Lattice Boltzmann Methods for Fluid Flows, Annu. Rev. Fluid Mech. 30: 329-364.

[12] Sukop M, Thorne D (2006). Lattice Boltzmann Modeling, 1st Edition. Springer.

[13] Qian YH, d'Humie`res D, Lallemand P (1992). Lattice BGK models for Navier–Stokes equation. Europhys. Lett. 17: 479-484.

[14] Mohamad AA, Kuzmin A (2010). A critical evaluation of force term in lattice Boltzmann method, natural convection problem. Internation Journal of Heat and Mass Transfer 53(5-6): 990-996.

[15] Feng ZG, Michaelides EE (2004). The immersed boundary-lattice Boltzmann method for solving fluid–particles interaction problems. J Comp Phy 195: 602–28.

[16] Cheng F, Zhang H (2010). Immersed boundary method and lattice Boltzmann method coupled FSI simulation of mitral leaflet flow. Computers & Fluid 39: 871-881.

[17] Peskin CS (2002). The immersed boundary method. Acta Numer 11: 479–517.

[18] Kuznik F, Obrecht C, Rusaouën G, Roux JJ (2009). LBM based flow simulation using GPU computing processor. Comput. Math. Appl. 59(7): 2380-2392.

[19] Lammers P, Küster U (2007). Recent Performance Results of the Lattice Boltzmann Method. High Performance Computing on Vector Systems 2006 Part 2, Springer, Stuttgart, 51-59.

[20] Kaminsky A (2015). Big CPU, Big Data. Rochester Institute of Technology.

[21] Fast Fluid-structure Interaction Using Lattice Boltzmann and Immersed Boundary Methods, M. Mawson, P. V. Lara, J. Favier, A. Pinelli, A. Revell

[22] Accelerating Solid-Fluid Interaction using Lattice-Boltzmann and Immersed Boundary Coupled. Simulations on Heterogeneous Platforms. Pedro Valero-Lara1, Alfredo Pinelli2, and Manuel Prieto-Matias3

[23] Von Karman T (1964). Aerodynamics, 1st Edition. McGraw-Hill.