J Supercomput (2014) 70:799–815
DOI 10.1007/s11227-014-1262-2

Accelerating solid–fluid interaction based on the immersed boundary method on multicore and GPU architectures

Pedro Valero-Lara

Published online: 12 July 2014
© Springer Science+Business Media New York 2014

Abstract This work proposes several approaches to accelerate the solid–fluid interaction through the use of the Immersed Boundary method on multicore and GPU architectures. Different optimizations on both architectures have been proposed, focusing on memory management and workload mapping. We have chosen two different test scenarios which consist of single-solid and multiple-solid simulations. The performance analysis has been carried out on an intensive set of test cases to analyze the proposed optimizations using multiple CPUs (2) and GPUs (4). An effective performance is obtained for single-solid executions using one CPU (Intel Xeon E5520) achieving a speedup peak equal to 5.5. It is reached a higher benefit on multiple solids obtaining a top speedup of approximately 5.9 and 9 using one CPU (8 cores) and two CPUs (16 cores), respectively. On GPU (Kepler K20c) architecture, two different approaches are presented as the best alternative: one for single-solid executions and one for multiple-solid executions. The best approach obtained for one solid executions achieves a speedup of approximately 17 with respect the sequential counterpart. In contrast, for multiple-solid executions the benefit is much higher, being this type of problems much more suitable for GPU and reaching a peak speedup of 68, 115 and 162 using 1, 2 and 4 GPUs, respectively.

Keywords Computational fluid dynamic · Parallel computing · Immersed boundary · Solid–fluid interaction · Multicore and GPU architectures

P. Valero-Lara (✉)
Numerical Simulation Unit, Research Center for Energy, Environment and Technology, CIEMAT, Madrid, Spain
e-mail: pedro.valero.lara@gmail.com; pedro.valero@ciemat.es

1 Introduction

The solid–fluid interaction is of vital importance for multiple computational fluid dynamics (CFD) applications with a high industrial and research interest. However, the inclusion of bodies within the fluid supposes a considerable overhead minimizing the performance of pure fluid solvers. This work focuses on the use of current parallel architectures to mitigate the cost that solid–fluid interaction supposes. The dynamics of a solid or a set of solids into a 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 about this topic reflects the growing importance of the study of the dynamics solid/s [1–7]. The use of current GPU architectures to compute the fluid field is widely extended into CFD due to the significant performance results achieved [8–13]. In contrast, the solid–fluid interaction has only recently gained wider topic.

Similar to the original work about the mathematical formulation of the immersed boundary (IB) algorithm presented by Peskin [14], we use an IB method based on the work of Uhlmann [15] to enforce the presence of a solid on the fluid field. Unlike other methods, this method is well established and has been used in numerous complex configurations, such as complex geometries, moving and deformable solids, etc, with satisfactory results [15–18]. We have focused on the optimization of this method due to the particular properties which IB presents, and the large range applications where it can be used. Furthermore, this is presented as an efficient, accurate and computationally cheap choice for this type of configurations.

Other authors address topics which are somehow related to the present contribution. López-Portugués et al. [19,20] propose the use of heterogeneous systems to compute the pressure distribution over the surface of one body basically solving Matrix–Vector product. Zhou et al. [13] show a solid–fluid solver based on curved boundary, where a flow around a circular cylinder is tested as a typical case. However, curved boundary is kept into account via a non-equilibrium extrapolation scheme. On the other hand, Laytona et al. [21] propose a GPU implementation based on the *IB projection* method [22] 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. Both previous works do not deal with multiple solids. Here, we will focus on a different approach based on *different forcing* approach [15] able to deal with complex, moving or deformable bodies, which has been used on Lattice Boltzmann [23] and Navier–Stokes [18] solvers.

This paper is structured as follows. Section 2 describes the general numerical framework of the IB method, based on the use of a set of singular forces distributed along the solid surfaces, and compares the experimental results obtained over other similar studies; Sect. 3 proposes a set of approaches to compute our problem on multicore and GPU architectures; Sect. 4 contains a performance analysis of the proposed approaches for single- and multiple-solid simulations; and finally, Sect. 5 outlines some final conclusions.

2 Mathematical formulation of the immersed boundary method

This section presents the mathematical formulation of the IB method. The basic idea consists of splitting the time advancement of the fluid momentum equation into two stage: the first without any body forces (no solid) and the second one adding to the right-hand side a body force restoring the zero velocity boundary condition on the solid surfaces. The core of IB consists of computing these body forces. The fluid is discretized on the regular Cartesian mesh, while the shape of the solids is discretized in a *Lagrangian* fashion by a set of points which obviously do not necessarily coincide with mesh points. This is sufficient information to impose the body forces on the solid surface.

It is necessary to compute these forces on a support, a set of Cartesian points around each *Lagrangian* (solid surface) point. The supports of continuous *Lagrangian* points of the same solid share several Cartesian points (Fig. 1). The key aspects of the algorithm are the interpolation $\mathcal{I}$ and the $\mathcal{S}$ operators (termed as spread from now on). Here, 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:

$$\mathbf{f}_{(ib)}(\mathbf{x},t) = \frac{1}{\Delta t}\int\limits_{\Gamma}\left(\mathbf{U}^{\mathbf{d}}(\mathbf{s},t+\Delta t) - \int\limits_{\Omega}\hat{\mathbf{u}}(\mathbf{y})\tilde{\delta}(\mathbf{y}-\mathbf{s})\,\mathrm{d}\mathbf{y}\right)\tilde{\delta}(\mathbf{x}-\mathbf{s})\,\mathrm{d}\mathbf{s} \tag{1}$$

where $\tilde{\delta}$ is the mollifier to be defined later, Γ is the set of *Lagrangian* points (immersed boundary), Ω is the computational domain, and $\mathbf{U}^{\mathbf{d}}$ is the desired value on the boundary at the next time step. The discrete equivalent of Eq. 1 is simply obtained by any

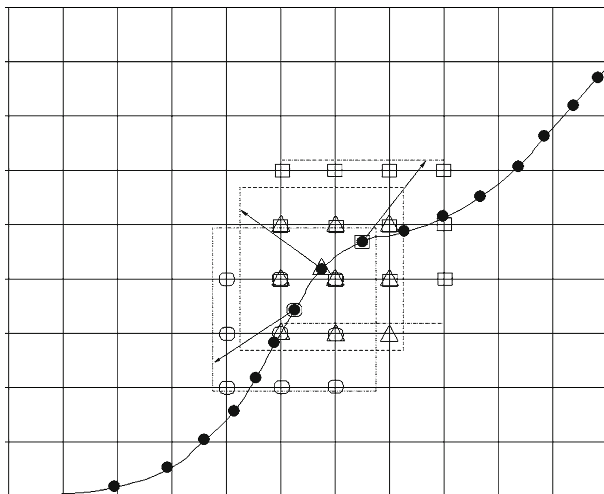


Fig. 1 An immersed curve discretized with *Lagrangian* points •. Three consecutive points are considered with the respective supports

standard composite quadrature rule applied on the union of the supports associated to each *Lagrangian* point. As an example, the quadrature needed to obtain the force distribution (spreading) on the Cartesian nodes is given by:

$$f^{ib}(x_i, y_j) = \sum_{n=1}^{N_e} F^{ib}(\mathbf{X_n})\tilde{\delta}(x_i - X_n, y_j - Y_n)\epsilon_n \tag{2}$$

(x_i, y_j) are the Cartesian nodes falling within the union of all the supports, N_e is the number of *Lagrangian* points and ϵ_n is a value to be determined to enforce consistency between interpolation and the spreading (Eq. 2). More details about the method and in particular about the determination of the ϵ_n values can be found in [18]. In what follows we will give more details on the construction of the support cages surrounding each *Lagrangian* point since it plays a key role in the parallel implementation of the IB method. As already mentioned, the solids surface are discretized into a number of *Lagrangian* points $\mathbf{X}_I,\ I = 1..N_e$. Around each point $\mathbf{X}_I$, we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes of the underlying Eulerian mesh for each direction; (ii) the number of nodes contained in the cage must be minimized. The modified kernel obtained as a Cartesian product of the one-dimensional function [24]:

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

will be identically zero outside the square Ω_I. We take the edges of the square to measure slightly more than three spacings Δ. With such choice, at least three nodes of the mesh in each direction fall within the cage. The interpolation stage is performed locally on each point support: the values of velocity at the nodes within the support cage centered about each *Lagrangian* point deliver approximate values (i.e., second order) of velocity at the point location. The force spreading step requires information from all the points. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring points, as in Eq. 2. Finally, the complete set of steps for the IB method is briefly described. Let the superscript * refers to the predicted variables without solids influence, $V_(C_)Lg^i_{x(y)}$ and $C_Sp^j_{x(y)}$ the horizontal (x) and the vertical (y) velocities (V), the coordinates (C) for the ith *Lagrangian* (Lg) point and the jth support (Sp) point:

1. **Compute the U^*_x and U^*_y fields without solid force**. These fields can be computed according to several algorithms as the solvers based on Navier–Stokes or Lattice Boltzmann.

2. **Velocities Interpolation (fluid → solid).**

$$
\begin{aligned}
V_Lg_x^i &+= \mathcal{I}(U_x^*[C_Sp_x^j]) \\
V_Lg_y^i &+= \mathcal{I}(U_y^*[C_Sp_y^j]) \\
&\forall i \in N_e, \forall j \in \Omega_i
\end{aligned}
\tag{4}
$$

3. **Compute the force on the solid surface (*Lagrangian* points).**

$$
\begin{aligned}
F_x^{ib}(\mathbf{X_i}) &= \mathbf{U_x^d} - V_Lg_x^i \\
F_y^{ib}(\mathbf{X_i}) &= \mathbf{U_y^d} - V_Lg_y^i \\
&\forall i \in N_e
\end{aligned}
\tag{5}
$$

4. **Spread the force (solid → fluid).**

$$
\begin{aligned}
f_x^{ib}(C_Sp_x^j, C_Sp_y^j) &+= \mathcal{S}(F_x^{ib}(\mathbf{X_i})) \\
f_y^{ib}(C_Sp_x^j, C_Sp_y^j) &+= \mathcal{S}(F_y^{ib}(\mathbf{X_i})) \\
&\forall i \in N_e, \forall j \in \Omega_i
\end{aligned}
\tag{6}
$$

5. **Adding the body forces to the U^* fields**. Depending of the method used to compute U^*, the f_{ib} is added.

$$
\begin{aligned}
U_x(C_Sp_x^j, C_Sp_y^j) &+= f_x^{ib}(C_Sp_x^j, C_Sp_y^j) \\
U_y(C_Sp_x^j, C_Sp_y^j) &+= f_y^{ib}(C_Sp_x^j, C_Sp_y^j) \\
&\forall i \in N_e, \forall j \in \Omega_i
\end{aligned}
\tag{7}
$$

Although the main contribution of this paper is the parallelization of the IB method, next we present several test cases to validate our implementation by comparing the numerical results obtained with other studies. One of the classical problems in CFD is the determination of the two-dimensional incompressible flow field around a circular cylinder, which is a fundamental problem in engineering applications. This problem has been studied in numerous works [1–4,13]. 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 conditions are set as: **Inlet**: $\mathbf{u} = U$, $\mathbf{v} = 0$, **Outlet**: $\partial\mathbf{u}/\partial x = \partial\mathbf{v}/\partial x = 0$, **Upper and lower boundaries**: $\partial\mathbf{u}/\partial y = 0$, $\mathbf{v} = 0$, **Cylinder surface**: $\mathbf{u} = 0$, $\mathbf{v} = 0$.

The numerical behavior became converged. 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 Fig. 2. Two important dimensionless numbers are studied, the drag ($CD = F_{\mathrm{D}}/0.5\rho U^2 D$) and lift ($CL = F_{\mathrm{L}}/0.5\rho U^2 D$) coefficients. Where F_{D} is the resistance and F_{L} is the lifting force of the circular cylinder, ρ is the density of the

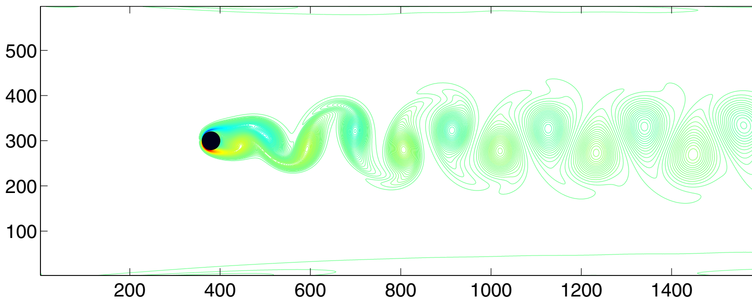


Fig. 2 Vorticity reached with $Re = 100$

Table 1 Experimental results

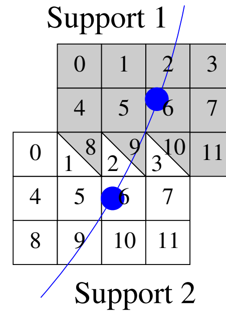
References	$Re = 20$	$Re = 40$	$Re = 100$	
	C_D	C_D	C_D	C_L
Calhoun [2]	2.19	1.62	1.33	0.298
Russell and Wang [3]	2.22	1.63	1.34	–
Silva et al. [4]	2.04	1.54	1.39	–
Xu and Wang [1]	2.23	1.66	1.423	0.34
Zhou et al. [13]	2.3	1.7	1.428	0.315
This work	2.3	1.7	1.39	0.318

fluid, and U is the velocity of inflow. 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. [4], and the lift coefficient is close to that presented by Zhou et al. [13].

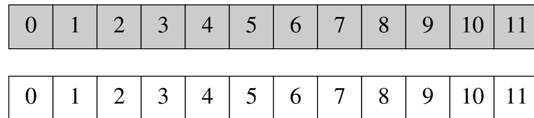
3 Parallel immersed boundary method for single and multiple solids

This section presents the strategy adopted to improve the solid–fluid interaction performance based on IB method using either multicore or GPU architectures.

It is well known that the memory management plays a crucial role in the performance of parallel computing. To compute the IB method, it is necessary to store the information about the coordinates, velocities and forces of all *Lagrangian* points and their supports. A set of memory management optimizations, which depends on the access pattern, has been carried out for the IB method implementation on both platforms, multicore and GPU, to achieve an effective memory usage. 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 multicore or GPU, since both architectures show different memory features and hierarchy.



No Coalescing – Sequential/Multicore



Coalescing – GPU

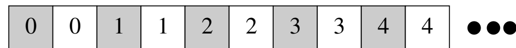


Fig. 3 Memory mapping on multicore and GPU architectures for two consecutive *Lagrangian* points

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, 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. For clarity, Fig. 3 shows the memory mapping performed on both platforms in a simplified example for two consecutive *Lagrangian* points.

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 and the size of theirs supports. 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 using the memory structures previously described (Fig. 3). The set of *Lagrangian* points can be easily parallelized with this approach, annotating some of its loops with Open-MP pragmas.

Concerning GPU, several approaches have been tested. The first approach (A) consists of exploiting the parallelism between *Lagrangian* points. All the steps are computed in only one kernel:

1. Velocities interpolation. The input parameters of this step are loaded from global memory using coalesced memory accesses. Its results are stored on local registers.
2. Force computation. The parameters are located in both, local and global memory (coalesced accesses). The results are located in local registers.

3. Spread the forces. The parameters are used from local and global memory and the results are stored in global memory using atomic operations.

Algorithm 1 A approach.

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

After the spreading step the forces are stored in the global memory 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 Fig. 1. Let us define a solid composed of "$numPoints$" *Lagrangian* points and "$numSupport$" support points per *Lagrangian* point. In this approach $BLOCKS_{USED} = numPoints/BLOCK_{SIZE}$ CUDA blocks are used, being $BLOCK_{SIZE}$ the number of threads per CUDA blocks. $BLOCKS_{USED} \times numSupport$ atomic operations are performed per CUDA block. Each thread loads and computes the information from $numSupport$ points. The pseudocode and the CUDA block-threads distribution are graphically illustrated in Algorithm 1 and Fig. 4, respectively.

On the other hand, our second GPU approach (B) consists of increasing the degree of parallelism by exploiting the independence between the support points, which is the maximum parallelism degree possible for our problem. In this case, a different strategy is proposed where the number of threads is equal to the number of all support points. To

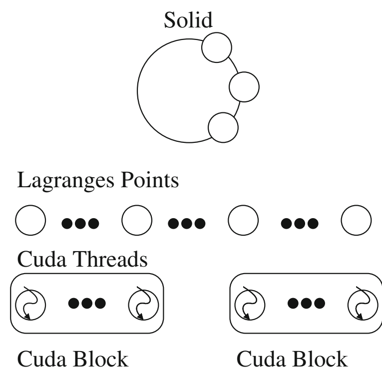


Fig. 4 CUDA block-threads distribution for the *A* approach

exploit the local memory in an efficient way, one CUDA block per *Lagrangian* point is used. However, to carry out the velocities interpolation on the support points and compute the force on the *Lagrangian* point, additional synchronizations points (barriers and atomic operations) and new reduction processes are necessary with respect to previous approach. In contrast, there is a lower computational cost per thread, where each thread computes and takes the parameters concerning only to one point of the support from global memory rather than taking the information of all support points. This approach is composed of the following steps:

1. Velocities interpolation. Each thread computes this step on a single support point. Both, parameters and results, are stored and accessed (coalescing accesses) on global memory. The results (velocities) are stored on global memory using atomic operations.
2. Synchronize threads.
3. Force computation. The first thread of each CUDA block carries out this step. The parameters are located in global memory and the results (forces) in local memory (shared memory).
4. Synchronize threads.
5. Spread the forces. The parameters are loaded from global (coalescing accesses) and local memory (forces). The results (forces on support points) are stored using atomic operations.

Algorithm 2 B approach.

1: B_approach(solid s,U_x,U_y)
2: **is** $(Threadandsupportpointid)$,**il** $(Lagrangianpointid)$
3: $Sharedforce_x$
4: $Sharedforce_y$
5: $AddAtom(s.XVelSupp[\mathbf{il}], interpol(U_x[s.Xsupp[\mathbf{is}]], s)$
6: $AddAtom(s.YVelSupp[\mathbf{il}], interpol(U_y[s.Ysupp[\mathbf{is}]], s)$
7: $SynchThreads$
8: **if** isupp == 0 **then**
9: $\quad force_x = computeForce(s.XVelSupp, s)$
10: $\quad force_y = computeForce(s.YVelSupp, s)$
11: **end if**
12: $SynchThreads$
13: $AddAtomic(s.XForceSupp, spread(force_x, s))$
14: $AddAtomic(s.YForceSupp, spread(force_y, s))$

This approach shows several disadvantages. The interpol step is computed on global memory using atomic operations instead of local memory to avoid incoherent executions, which difficulties the memory management over the previous approach. Therefore, for the same case as the previous approach $numPoints \times numSupport$ CUDA blocks are used, each thread has a lower computational cost and $numPoints \times numSupport$ additional atomic operations and global memory accesses are carried out. Algorithm 2 shows the differences with the previous approach and Fig. 5 illustrates graphically the CUDA blocks mapping.

Other GPU approach (C) is proposed to mitigate the disadvantages suffered by the previous approach. Instead of using atomic operations in the “interpol” step, a

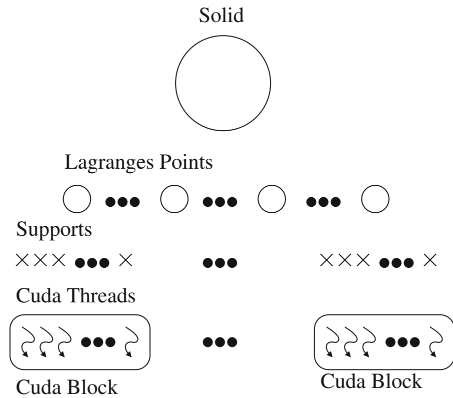


Fig. 5 CUDA threads distribution for the *B* and *C* approaches

reduction strategy is adopted where the first thread of each CUDA block takes the velocities of all support points (shared memory), which are previously computed by all threads of the CUDA block, and computes the final velocity on *Lagrangian* point. The same strategy is followed in the "compute the force" step. In this way, only one atomic operation per support point is used, as in the case of *A* approach. The steps carried out in this approach are:

1. Velocities interpolation. The result is stored in shared memory and the parameter is taken from global memory (coalescing accesses).
2. Synchronize threads.
3. Gather the local velocities. The first thread of each CUDA block adds all local velocities. Both, parameters and results, are located in local memory.
4. Force computation. The first thread of each CUDA block carries out this step. The parameters are located in global and local memory (velocities) and the results (forces) in local memory (shared memory).
5. Synchronize threads.
6. Spread the forces. The results are stored on global memory using atomic operation, and the parameters are taken from local and global memory.

It is important to note that the reduction strategy adopted by the *B* and *C* during the step "compute the forces" increases the synchronization points and suffers from divergence in the execution path. Although, there are parallel techniques which allow to carry out reduction processes, these do not obtain better performance than the proposed strategy. This is due to the small size of the supports. Both approaches share the same number of CUDA blocks ($numPoints \times numSupport$) and the same CUDA blocks mapping (Fig. 5). In contrast, the *C* approach does not present an additional overhead caused by the atomic operation and accesses to global memory with respect to the *A* approach. The main features about this approach are detailed in Algorithm 3.

A final approach (*D*) is proposed for the particular case of multiple-solid executions. This approach is similar to *A* approach, however, instead of using a set of CUDA blocks to carry out the IB on one solid, one CUDA block per solid is used. In addition, the memory hierarchy can be efficiently exploited using this approach, since this configuration (multiple solids) is much more profitable for GPU computing. Concerning

Algorithm 3 SUPPORT-RED approach.

1: SUPPORT-RED_approach(solid s,U_x,U_y)
2: **is** $(Threadandsupportpointid)$
3: $Sharedvel_x[numSupp]$, $vel_y[numSupp]$, vel_{xtotal}, vel_{ytotal}, $force_x$, $force_y$
4: vel_x[**is**] $= interpol(Vel_x[solid.Xsupp$[**is**]$], s)$
5: vel_y[**is**] $= interpol(Vel_y[solid.Xsupp$[**is**]$], s)$
6: $SynchThreads$
7: **if** isupp == 0 **then**
8:     **for** $i = 1 \rightarrow numSupport$ **do**
9:         $vel_{xtotal} += vel_x[i]$
10:         $vel_{ytotal} += vel_y[i]$
11:     **end for**
12:     $force_x = computeForce(vel_{xtotal}, s)$
13:     $force_y = computeForce(vel_{ytotal}, s)$
14: **end if**
15: $SynchThreads$
16: $AddAtomic(s.XForceSupp, spread(force_x, s))$
17: $AddAtomic(s.YForceSupp, spread(force_y, s))$

to multicore, a different approach is proposed to compute multiple solids as well. Similar to the previous approach, the only difference is identified in the workload mapping a set of solids per core instead of a set of *Lagrangian* points per core.

4 Performance analysis

This section provides a performance analysis considering all the IB approaches, sequential (single core), multicore, and GPU. All computations are carried out using double precision. GCC optimization flags (-O3, -funroll-loops) are used for sequential and multicore approaches. The results achieved in this section are shown in terms of execution time and speedup. Speedup is the ratio between the execution time obtained by the sequential over the parallel counterpart. The results in term of execution time (the average per interaction) are shown in tables, whereas the results in terms of speedup are shown in figures.

We have used a heterogeneous CPU–GPU platform, composed of the computational resources shown in Table 2. The overhead caused by the memory transfer between

Table 2 Platform

Platform	Xeon E5520 (2.26 GHz)	Kepler K20c
Cores	8	2,496
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	64 GB DDR3	5 GB GDDR5
Bandwidth (GB/s)	51.2	208
Compiler	gcc 4.6.2	nvcc 5.5

memories is included on the results obtained by the GPU approaches. The analysis consists of two subsection: one for single solid and one for multiple solids. The set of tests consisted in obtaining the dynamics of a cylinder/s immersed into the fluid. The size of the cylinder/s (radius) is increased to evaluate the trend in performance for higher computational requirements. Given the radius of the cylinder, the number of *Lagrangian* points is calculated using floor($2 \times$ radius $\times \pi$).

4.1 Single solid

As shown (Table 3; Fig. 6), the multicore approach achieves an efficient and constant performance of around 5.5 in terms of speedup using 8 cores. The *A* approach presents better results with respect to the multicore counterpart from a radius equal to 128, although this benefit is not higher than the rest of the GPU approaches. The *B* approach

Table 3 Execution time (ms) consumed by the set of IB approaches for single-solid executions

Radius	IB approaches				
	1 Th	8 Th	A	B	C
8	0.218	0.0419	0.233	0.069	0.062
16	0.254	0.0492	0.235	0.072	0.068
32	0.527	0.0958	0.295	0.076	0.071
64	1.05	0.19	0.301	0.09	0.082
128	2.14	0.389	0.322	0.147	0.127
256	3.97	0.721	0.405	0.276	0.235
512	8.06	1.468	0.526	0.567	0.476
1,024	15.73	2.84	0.972	1.11	0.924
2,048	28.95	5.18	1.782	2.06	1.71

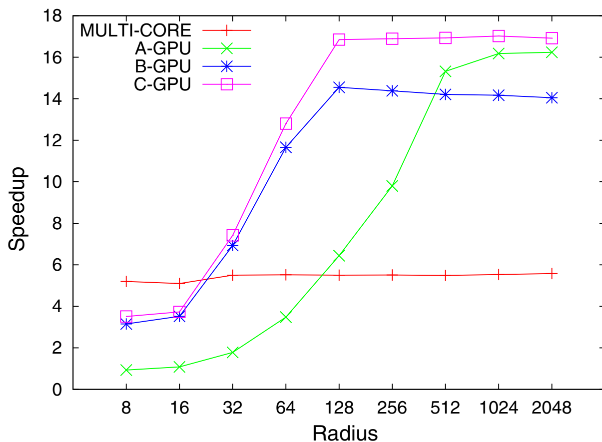


Fig. 6 Trend of the speedup reached by the set of IBM approaches increasing the size (radius) of the solid

shows a different behavior and offers better results. However, the performance reached by the *C* approach is even better, since this approach does not present the disadvantages shown by the *B* approach caused by a higher number of atomic operations and global memory accesses. This overhead became higher as the size of the cylinder increases. A peak of performance of approximately 14 and 17, in terms of speedup, is obtained by the *B* and *C* approaches, respectively. In contrast, the *A* approach does not saturate the GPU performance up to a radius equal to 512 reaching a top speedup around 16. It is necessary to increase the size of the solid up to a minimum radius equal to 32 to achieve the sufficient workload and parallel features to obtain better performance over the multicore counterpart on both approaches, *B* and *C*.

4.2 Multiple solids

This problem shows higher computational requirements and parallel potential so that we have taken into account the use of multiple CPUs (2) and GPUs (4). Four different approaches have been tested: the sequential, the multicore applied to multiple solids, the *C* and *D* approaches on GPUs. For single-solid simulations, the *A* approach does not provide satisfactory results compared with the other GPU approaches and the *C* approach is able to achieve the best performance. We want to know if this is still valid for multiple-solid simulations (Table 4).

In the previous subsection the only parameter evaluated was the size of the cylinder (radius). However, in this case, the parallelism degree is given by the number of cylinders as well. Several tests were carried out by increasing the radius and number of solids to determine the influence on performance of both parameters.

The trend in performance on multicore architecture (Fig. 7) is very similar to that of the previous subsection. However, the different mapping of the workload obtains a higher benefit achieving a speedup of nearly 5.9 using 1 CPU (8 cores). Furthermore, an efficient and constant speedup is obtained using 2 CPUs (16 cores) with a speedup nearly 9.

The results obtained by both GPU approaches show much better performances. As shown in Fig. 8, the *C* approach achieves a performance peak of approximately 35 even in the first tests cases (low number of cylinders). This demonstrates that this approach does not scale for multiple-solid simulations. In contrast, the *D* approach offers better performances, achieving a peak of performance of approximately 68. The gain improves as the number of cylinder and radius increases until the maximum benefit is reached. However, for small solids and few number of solids the *C* approach continues being the best choice.

There is no additional overhead to compute the IB on multiple GPUs, since the solving of each solid is totally independent. On both multiGPU approaches (2 and 4 GPUs) the trend reached (Fig. 9) is very similar to that of the use of 1 GPU. The use of multiple GPUs reaches a higher speedup over the use of one GPU from 6.89 and 7.38 in the first test case (8 solids/radius) to 118 and 164 in the last test case (256 solids/radius) using 2 and 4 GPUs, respectively.

Table 4 Execution time (ms) achieved by the set of IB approaches for multiple-solid executions

#Sol.	IB approaches						
	1 Th	8 Ths	16 Ths	C App.	D App.	2 GPUs	4 GPUs
Radius = 8							
8	1.64	0.282	0.188	0.095	0.244	0.238	0.222
16	2.61	0.453	0.296	0.141	0.272	0.253	0.24
32	5.43	0.932	0.617	0.26	0.289	0.277	0.262
64	11.9	2.04	1.37	0.341	0.313	0.292	0.281
128	19.2	3.29	2.16	0.492	0.389	0.319	0.295
256	38.9	6.84	4.23	0.992	0.787	0.625	0.457
Radius = 16							
8	2.43	0.422	0.279	0.141	0.267	0.252	0.232
16	5.84	1	0.675	0.259	0.307	0.27	0.262
32	10.1	1.72	1.14	0.442	0.33	0.313	0.273
64	15.4	2.64	1.74	0.505	0.374	0.339	0.321
128	21.7	3.71	2.44	0.903	0.437	0.385	0.342
256	41.3	7.01	4.49	1.7	0.801	0.705	0.583
Radius = 32							
8	5.41	0.929	0.629	0.263	0.315	0.297	0.242
16	11	1.89	1.27	0.344	0.348	0.322	0.308
32	13.6	2.33	1.56	0.503	0.384	0.356	0.331
64	23.7	4.03	2.66	0.931	0.505	0.485	0.371
128	49.3	8.39	5.58	1.66	0.826	0.771	0.585
256	87.4	14.94	9.87	3.02	1.41	1.16	0.647
Radius = 64							
8	8.66	1.48	0.974	0.344	0.352	0.343	0.297
16	11.9	2.05	1.34	0.496	0.395	0.365	0.355
32	24.3	4.16	2.72	0.928	0.74	0.401	0.370
64	49.8	8.78	5.61	1.65	1.02	0.83	0.423
128	99.4	17.3	11.1	3.11	1.6	1.28	0.89
256	159.9	28.01	17.84	5.01	3.1	1.81	1.08
Radius = 128							
8	11.8	2.03	1.32	0.496	0.403	0.392	0.356
16	23.9	4.09	2.68	0.932	0.751	0.412	0.403
32	50.5	8.67	5.66	1.66	1.05	0.807	0.432
64	98.7	16.8	11	3.13	1.63	1.19	0.815
128	200	34	22.49	6.45	2.94	1.73	1.23
256	389.3	65.82	42.79	12.36	5.82	3.32	2.387

Table 4 continued

	IB approaches						
#Sol.	1Th	8Ths	16Ths	C App.	D App.	2GPUs	4GPUs
Radius = 256							
8	24.4	4.21	2.71	0.911	0.769	0.428	0.389
16	53.21	9.08	5.78	1.69	1.06	0.887	0.442
32	99.01	17.23	10.84	3.22	1.68	1.23	0.819
64	197.97	34.07	22.47	6.41	2.96	1.78	1.35
128	371.56	65.31	41.02	12.03	5.46	3.2	2.28
256	610.77	107.3	68	19.58	9.02	5.25	3.71

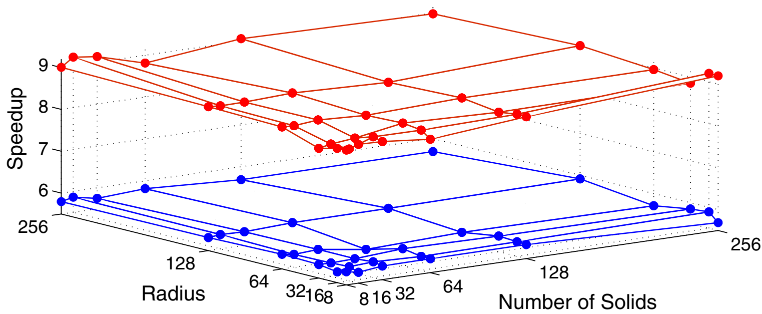


Fig. 7 Trend of the speedup obtained by the multicore approach using 2 CPUs-16 cores (*top*) and 1 CPU-8 cores (*bottom*) increasing the radius and the number of solids

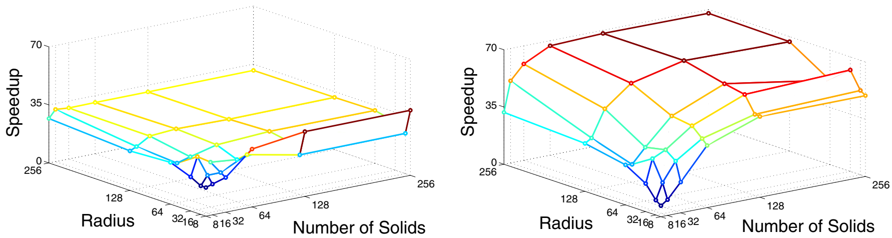


Fig. 8 Speedup for the *C* (*left*) and *D* (*right*) approaches increasing the radius and the number of solids

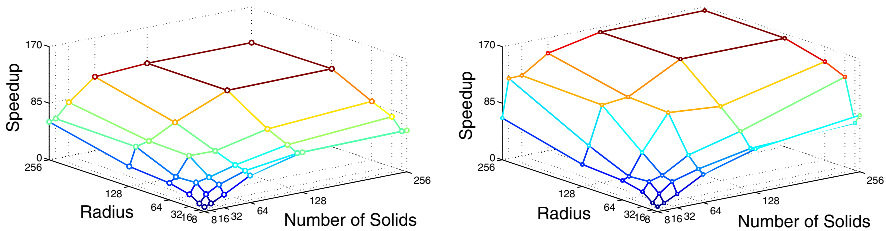


Fig. 9 Speedup for the multiple-GPU approaches using two (*left*) and four GPUs (*right*)

5 Conclusions

The immersed boundary method has been used to simulate the solid–fluid interaction on multicore and GPU architectures. A classical test which consists of computing the flow around single or multiple cylinders has been used with satisfactory results in terms of accuracy and efficiency. Several approaches for the implementation of the IB method have been proposed. The memory management has been optimized for both architectures, adapting the use of memory to the particularities of each memory hierarchy.

Efficient speedups for single- and multiple-solid executions have been achieved by exploiting the multicore architecture, demonstrating the efficiency of the memory management and the implementation. The different CUDA blocks mappings, one CUDA block per support point (B and C approach), one CUDA block per *Lagrangian* point (A approach) and one CUDA block per solid (D approach), have a direct consequence on performance. The highest performance in terms of speedup and execution time for single-solid executions has proven to be the C approach, which combines the level of parallelism presented by the B approach with the low number of global memory accesses and atomic operations of the A approach. However, it is required a minimum radius equal to 32 for one solid simulations to reach sufficient workload and parallel features to achieve better results with respect to the multicore counterpart. On the other hand, the D approach offers the best performance for multiple-solid simulations achieving better results in all test cases compared with the multicore approach. Concerning to the use of multiple GPUs, a high computational load (number of solids and radius) is required to achieve a significant benefit over the use of one GPU.

Acknowledgments This work has been supported by the Spanish Consolider grant Supercomputación y e-Ciencia (SyeC) (Ref: CSD2007-00050) and by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT) [25]. The author would like to thank Julien Favier, Assistant Professor at Aix Marseille Université, Alfredo Pinelli at Full Professor at City of London University and Manuel Prieto-Matias at Complutense University of Madrid, for the valuable feedback on this work.

References

1. Xu S, Wang ZJ (2006) An immersed interface method for simulating the interaction of a fluid with movies boundaries. J Comput Phys 216(2):454–493
2. Calhoun D (2002) A Cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. J Comput Phys 176(2):231–275
3. Russell D, Wang ZJ (2003) A Cartesian grid method for modelling multiple moving objects in 2D incompressible viscous flows. J Comput Phys 191:177–205
4. Lima E Silva ALF, Silveira-Neto A, Damasceno JJR (2003) Numerical simulation of two-dimensional flows over circular cylinder using immersed boundary method. J Comput Phys 189:351–370
5. Tutar M, Holdo AE (2001) Computational modelling of flow around cylinder in sub-critical flow regime with various turbulence models. Int J Numer Methods Fluids 35(7):763–784
6. Norbeg C (2003) Fluctuating lift on a circular cylinder: review and new measurements. J Fluids Struct 17(1):57–96
7. Ong L, Wallace J (1996) The velocity field of a turbulent very near wake of a circular cylinder. Exp Fluids 20(6):441–453
8. Goodnight N (2011) CUDA/OpenGL fluid simulation. http://new.math.uiuc.edu/MA198-2008/schamber2/fluidsGL

9. Tolke J (2010) Implementation of a Lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA. Comput Visual Sci 13(1):29–39
10. Zhao Y (2007) Lattice Boltzmann based PDE solver on the GPU. Vis Comput 24(5):323–333
11. Bernaschi M, Fatica M, Melchiona S, Succi S, Kaxiras E (2010) A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. Concurrency Computa Pract Exper 22:1–14
12. Rinaldi PR, Dari EA, Vénere MJ, Clausse A (2012) A Lattice–Boltzmann solver for 3D fluid simulation on GPU. Simul Model Pract Theory 55:163–171
13. Zhou H, Mo G, Wu F, Zhao J, Rui M, Cen K (2012) GPU implementation of lattice Boltzmann method for flows with curved boundaries. Comput Methods Appl Mech Eng, 225–228
14. Peskin CS (2002) The immersed boundary method. Acta Numer 11:479–517
15. Uhlmann M (2005) An immersed boundary method with direct forcing for the simulation of particulate flows. J Comput Phys 209(2):448–476
16. Zhu L, Peskin CS (2000) Interaction of two flapping filament in a flow soap film. Phys Fluids 15:1954–1960
17. Zhu L, Peskin CS (2002) Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method. Phys Fluids 179:452–468
18. Pinelli A, Naqavi I, Piomelli U, Favier J (2010) Immersed-Boundary methods for general finite-differences and finite-volume Navier–Stokes solvers. J Comput Phys 229(24):9073–9091
19. L'opez-Portugués M, López-Fernández JA, José Ranilla, Ayestarán RG, Las-Heras F (2013) Parallelization of the FMM on distributed-memory GPGPU systems for acoustic-scattering prediction. J Supercomput 64(1):17–27
20. López-Portugués M, López-Fernández JA, Díaz-Gracia N, Ayestarán RG, José Ranilla (2014) Aircraft noise scattering prediction using different accelerator architectures, pp 1–11
21. Laytona SK, Krishnana A, Barbaa LA (2011) cuIBM—a GPU-accelerated immersed boundary method. 23rd international conference on parallel computational fluid dynamics (ParCFD)
22. Taira K, Colonius T (2007) The immersed boundary method: a projection approach. J Comput Phys 225(2):2118–2137
23. Favier J, Revell A, Pinelli A (2013) A lattice boltzmann-immersed boundary method to simulate the fluid interaction with moving and slender flexible objects. HAL hal (00822044)
24. Roma AM, Peskin CS, Berger MJ (1999) An adaptive version of the immersed boundary method. J Comput Phys 153:509–534
25. CETA-CIEMAT. http://www.ceta-ciemat.es

Copyright of Journal of Supercomputing is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.