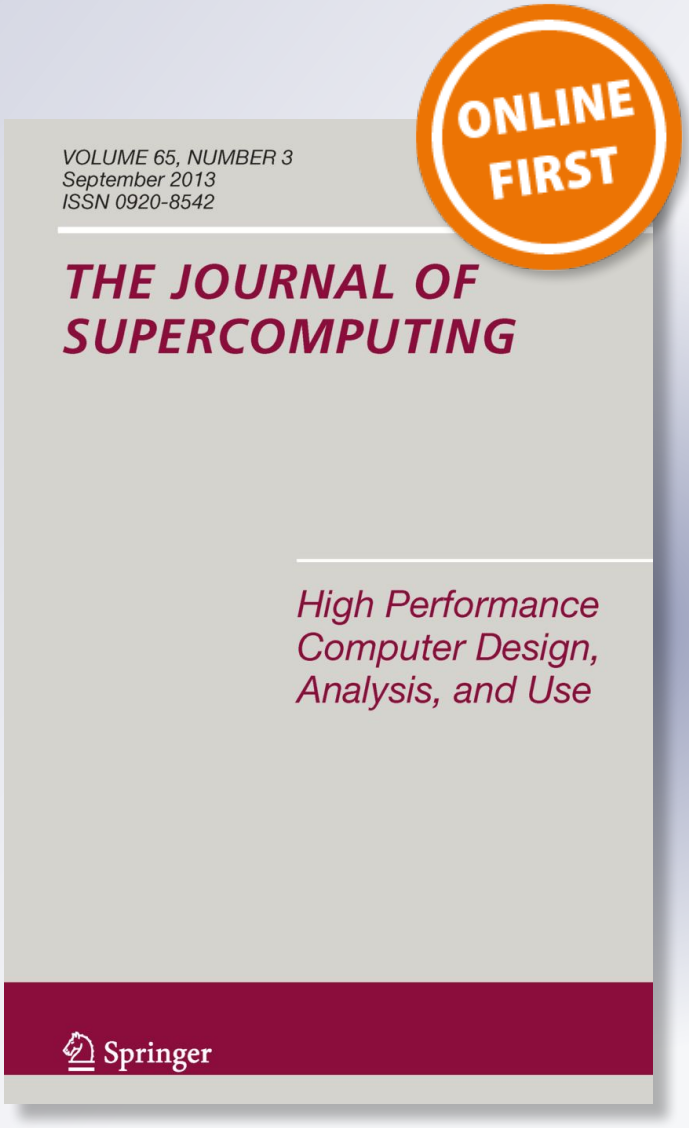
Interactive 3D simulation for fluid–structure interactions using dual coupled GPUs

Bob Zigon, Luoding Zhu & Fengguang Song

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-017-2103-x



Springer

J Supercomput
DOI 10.1007/s11227-017-2103-x

Interactive 3D simulation for fluid–structure interactions using dual coupled GPUs

Bob Zigon[1] · **Luoding Zhu**[2] · **Fengguang Song**[3]

Abstract The scope of this work involves the integration of high-speed parallel computation with interactive, 3D visualization of the lattice-Boltzmann-based immersed boundary method for fluid–structure interaction. An NVIDIA Tesla K40c is used for the computations, while an NVIDIA Quadro K5000 is used for 3D vector field visualization. The simulation can be paused at any time step so that the vector field can be explored. The density and placement of streamlines and glyphs are adjustable by the user, while panning and zooming is controlled by the mouse. The simulation can then be resumed. Unlike most scientific applications in computational fluid dynamics where visualization is performed after the computations, our software allows for real-time visualizations of the flow fields while the computations take place. To the best of our knowledge, such a tool on GPUs for FSI does not exist. Our software can facilitate debugging, enable observation of detailed local fields of flow and deformation while computing, and expedite identification of 'correct' parameter combinations in parametric studies for new phenomenon. Therefore, our software is expected to shorten the 'time to solution' process and expedite the scientific discoveries via scientific computing.

✉ Bob Zigon
robert.zigon@beckman.com

1 Beckman Coulter, Indianapolis, IN 46268, USA

2 Department of Mathematical Sciences, Indiana University-Purdue University Indianapolis, Indianapolis, IN 46202, USA

3 Department of Computer Science, Indiana University-Purdue University Indianapolis, Indianapolis, IN 46202, USA

Published online: 19 July 2017

Keywords Lattice Boltzmann method · The immersed boundary method · GPU computing · Fluid–structure interaction(FSI) · Interactive simulation · Real-time visualization

1 Introduction

The fluid–structure interaction (FSI) problem involves the interplay of a solid structure with a surrounding fluid flow. Such problems are ubiquitous and include examples like the deformation of a fish's fin while swimming [1], the bending of the cilium in the kidney lumen in response to shear flow [2], or the strong wind effect on a skyscraper [3]. Because the interaction of an elastic solid and a viscous fluid in nature is nonlinear, multi-physics, and multi-scale, an analytical solution is very rare. Instead, a computational approach is practically viable.

To date, many computational methods exist for numerical studies of problems involving flow–structure interactions, probably because of the complexity and diversity of real-world FSI problems and the limitations of mathematics and computer resources. Each method has its strength and weakness. Some of them are comparable. The choice is problem dependent. We shall not attempt to compare these methods here; instead we list some of them for readers' reference. These methods include immersed boundary (IB) methods [4,5], the immersed interface methods [6], blob-projection [7], immersed continuum [8], and immersed finite element [9], the Arbitrary Lagrangian Eulerian (ALE) [10], the fictitious domain method [11], the material point method [12], the level set method [13], and the front tracking method [14].

We choose to use the popular IB method originated by Peskin [15] for the fluid–structure interactions. The reason is that it is well tested, efficient, and allows a variety of fluid and solid solvers to be combined. Within the IB method, there too exists different versions. Examples include the original versions [16], the vortex-method version [17], the volume-conserved version [18], the adaptive mesh refinement version [19], the (formally) second-order versions [20,21], the multigrid version [22], the penalty version [23], the implicit versions [24–28], the generalized version for a thick rod [29], the stochastic version [30], the porous media version, the lattice-Boltzmann version [31–41], the fluid-solute-structure interaction version [42], and the variable viscosity version [43].

In this work, we strive to realize the *lattice-Boltzmann-based immersed boundary* (*LB-IB*) method developed by Zhu et al. [31] for general fluid-deformable-structure interactions. The lattice Boltzmann (LB) method [44–51] is a widely used alternative to traditional numerical methods for flow problems. It employs a meso-scale description and incorporates a velocity distribution function that obeys an approximate Boltzmann equation. Compared to conventional approaches for solving the flow problem, the LB method is relatively simpler to use, easier to handle complex rigid boundaries (e.g., porous media), and more convenient to incorporate additional physics into a model to simulate new flow phenomena, particularly in three dimensions. Therefore, our software can be easily extended to other situations such as FSI involving non-Newtonian fluids.

Another reason for choosing the LB-IB method is the inherent parallelism in both the lattice Boltzmann (LB) method and the immersed boundary method, which makes them good candidates for parallel computing on GPUs [52]. In the LB method, with each node in the computing domain acting independently of its neighbors, the streaming and collision of the fluid particles maps elegantly on to the thousands of cores present on a GPU. In a similar manner, in the IB method, the computation of the forces and subsequent application to the immersed object is also a procedure where adjacent domain members can be computed in parallel.

The literature shows the existence of some work implementing the lattice Boltzmann and immersed boundary methods on GPUs. Valero et al. [53] demonstrated the performance of the 2D algorithm on Intel CPU's and NVIDIA GPU's. The goal of their work was to investigate optimization strategies for heterogeneous architectures on two-dimensional domains. In the work of Mawson et al. [54], they too developed a GPU library. Like Valero, they focused on implementation and performance, but also investigated the application to 3D domains. Although Mawson et al. stated that real-time 3D simulations are possible with GPU acceleration, they also found it was very difficult to identify the z-depth for the object they placed in the field. They did not perform visualization, nor did they address how complex internal obstacles could be handled in their work. In 2016, Wu [55] developed a GPU accelerated LB-IB simulator for a three-dimensional ellipsoidal membrane. They focused on the creation of efficient code for computation on a single GPU. This is in contrast to our approach which distributed the computation and visualization across two distinct GPUs.

Our work explores the use of GPUs not only to accelerate computation, but also interactively visualize computational results in real time in three dimensions. We have created a number of C++ classes that simplify the implementation on GPUs. To compare and demonstrate the efficiency of using the GPU, we also implement the algorithms on CPUs using OpenMP. We model a 3D viscous flow past a deformable mesh fixed at its midline behind a circular rigid cylinder as an example of our work; however, our software implementation is generic and can be used for other FSI problems.

Visualization has become an essential part in engineering, research and business workflows. The current practice for a researcher in computational fluid dynamics is as follows: One executes a simulation and saves the data to storage, waits for the simulation to terminate, and then loads the data from storage for visualization. For large-scale real-world FSI problems, a simulation may take days or even weeks to finish on modern parallel computers. Furthermore, computational studies frequently perform many series of simulations with different combinations of problem-specific parameters (i.e., *parametric studies*). Due to the essential *nonlinearity* of FSI problems, the "correct" choices that may lead to new phenomenon or discovery are typically not known beforehand. The real-time visualization may substantially help in this regard by identifying the uninteresting or incorrect combinations long before simulations are completed. For this reason, we have made it easy for the researcher to rotate and zoom the simulation during the computation. We also allow the user to view the vector field using three-dimensional vectors or streamlines, as well as changing their density and placement, to better observe the behavior of the flow.

Integrating visualization into the computing framework brings value in other areas. A domain with 256^3 nodes that simulates for 10^5 s would consume 12 terabytes of disk

space if the simulation was saved. With our approach, you simply rerun the simulation. Moreover, any disk I/O that is incurred during simulation would certainly slow down the overall execution time if every time step is written to disk. We have also found that the interactivity simplified debugging. When a logic error occurred, the source was frequently very obvious from the output.

Regarding visualization, software toolkits like the OpenGL Volumizer [56], ParaView [57] and VisIt [58] enable a user to interactively visualize the data after it has been computed. In our work, we integrate large-scale simulation with real-time visualization using two GPUs. An NVIDIA Tesla K40c is used for computations, while an NVIDIA Quadro K5000 is used for 3D visualization by streamlines or vector glyphs.

To the best of our knowledge, this paper makes the following three contributions. First, we present an efficient GPU implementation of the LB-IB method in three dimensions. Second, we create a set of software classes capable of supporting both CPUs and GPUs. Finally, we provide an integrated approach to realizing online FSI visualization using multiple GPUs that emphasizes human interaction with the simulator during computation. This capability facilitates code debugging, allows one to observe detailed local flow and deformation dynamics while computing, and expedites identification of 'correct' parameter combinations in parametric studies for new phenomenon. It is also expected to shorten the 'time to solution' process and expedite the scientific discoveries via scientific computing.

The remainder of the paper is organized as follows. Section 2 introduces the LB-IB method, including the mathematical formulation and its discretization. Section 3 describes the software design for the OpenMP and GPU hardware platforms. Section 4 gives implementation details. Section 5 presents our results, and Sect. 6 concludes with a discussion of future work.

2 The LB-IB method

2.1 The mathematical formulation

The lattice Boltzmann method originated from Boltzmann's kinetic theory of dilute gases. The fundamental concept is that fluids can be modeled as large collections of particles with random motions. The exchange of momentum and energy is achieved through particle collisions and particle streaming. The LB method is an alternative to traditional numerical methods such as the fast Fourier Transform, the projection method, and the particle in cell method for obtaining the solution to the viscous incompressible flow problem. In contrast to solving for macroscopic variables like velocity and pressure, the LB method uses a mesoscopic approach that deals with a particle velocity distribution function $g(\mathbf{x}, \xi, t)$ defined on a Eulerian grid. Here $\mathbf{x}$ represents the spatial coordinate, ξ represents particle velocity, and t is time.

Different from the lattice Boltzmann method, the goal of the immersed boundary method is to model the interaction of a fluid with an elastic material. The elastic material is treated as part of the fluid in which additional forces are applied. The elastic material is tracked on a Lagrangian grid by following the material points. The

configuration of these points is used to compute elastic forces which are applied to the nearby lattice points of the fluid.

Our overall approach for the LB-IB formulation follows that of [31]. The dimensionless form is formulated as follows:

$$\frac{\partial g(\mathbf{x},\xi,t)}{\partial t}+\xi\cdot\frac{\partial g(\mathbf{x},\xi,t)}{\partial \mathbf{x}}+\mathbf{f}_{ib}(\mathbf{x},t)\cdot\frac{\partial g(\mathbf{x},\xi,t)}{\partial \xi}=-\frac{1}{\tau}(g(\mathbf{x},\xi,t)-g^{(0)}(\mathbf{x},\xi,t)). \tag{1}$$

Bhatnagar–Gross–Krook (BGK) [59] are attributed to Eq. (1) which describes the motion of both the fluid and the immersed boundary in the context of the LB method. The quantity $g(\mathbf{x},\xi,t)\,\mathbf{dx}\,\mathbf{d}\xi$ represents the probability of finding a particle at time t, located in the interval $[\mathbf{x}, \mathbf{x}+\mathbf{dx}]$, while moving with velocity in the interval $[\xi, \xi+\mathbf{d}\xi]$. The term

$$-\frac{1}{\tau}(g(\mathbf{x},\xi,t)-g^{(0)}(\mathbf{x},\xi,t)) \tag{2}$$

in (1) is the BGK approximation to the complex collision operator in the Boltzmann equation, where τ is the relaxation time and $g^{(0)}(\mathbf{x},\xi,t)$ is the Maxwellian distribution. The term $\mathbf{f}_{ib}(\mathbf{x},t)$ is the force imparted by the immersed boundary to the fluid. This term is largely responsible for the unification of the LB and IB methods. As a result, there is no need to explicitly remesh the immersed boundary because the two methods are coupled by way of $\mathbf{f}_{ib}(\mathbf{x},t)$.

The LB method requires the macroscopic variables fluid mass density, $\rho(\mathbf{x},t)$, and the momentum, $(\rho\mathbf{u})(\mathbf{x},t)$, which are defined in (3) and (4) as functions of the velocity distribution function $g(\mathbf{x},\xi,t)$.

$$\rho(\mathbf{x},t)=\int g(\mathbf{x},\xi,t)\,d\xi \tag{3}$$

$$(\rho\mathbf{u})(\mathbf{x},t)=\int g(\mathbf{x},\xi,t)\xi\,d\xi \tag{4}$$

The Eulerian force density $\mathbf{f}_{ib}(\mathbf{x},t)$ defined on the fixed Eulerian lattice is calculated from the Lagrangian force density $\mathbf{F}_{ib}(\boldsymbol{\alpha},t)$ defined on the Lagrangian grid by Eq. (5),

$$\mathbf{f}_{ib}(\mathbf{x},t)=\int \mathbf{F}_{ib}(\boldsymbol{\alpha},t)\delta(\mathbf{x}-\mathbf{X}(\boldsymbol{\alpha},t))\,d\boldsymbol{\alpha} \tag{5}$$

where the function $\delta(\mathbf{x})$ is the Dirac δ-function. The Lagrangian force density $\mathbf{F}_{ib}$ is computed as follows:

$$\mathbf{F}_{ib}(\boldsymbol{\alpha},t)=-\frac{\partial\mathcal{E}}{\partial\mathbf{X}}=-\frac{\partial(\mathcal{E}_s+\mathcal{E}_b)}{\partial\mathbf{X}} \tag{6}$$

In Eq. (6), the elastic potential energy density $\mathcal{E}$ consists of a stretching/compression component $\mathcal{E}_s$ and a bending component $\mathcal{E}_b$. These last two quantities are defined by Eqs. (7) and (8), respectively.

$$\mathcal{E}_s=\frac{1}{2}K_s\int\int d\alpha_2\,d\alpha_3\int\left(\left|\frac{\partial\mathbf{X}(\boldsymbol{\alpha},t)}{\partial\alpha_1}\right|-1\right)^2 d\alpha_1 \tag{7}$$

$$\mathcal{E}_b = \frac{1}{2} K_b \int \int d\alpha_2 \, d\alpha_3 \int \left(\left| \frac{\partial^2 \mathbf{X}(\boldsymbol{\alpha}, t)}{\partial \alpha_1^2} \right| \right)^2 d\alpha_1 \tag{8}$$

The variables $\alpha_1, \alpha_2, \alpha_3$ are the three components of the Lagrangian variable $\boldsymbol{\alpha}$. In the case of an immersed surface, such as the flexible membrane in Sect. 5, α_2 may be used to denote a fiber, α_1 to denote the arc length along the fiber, and α_3 is not used. K_s is the stretching/compression coefficient, and K_b is the bending coefficient. Both constants are related to Young's modulus of the membrane.

The motion of the flexible membrane is described by a system of first-order ordinary differential equations. Equation (9) describes the system.

$$\frac{\partial \mathbf{X}}{\partial t}(\boldsymbol{\alpha}, t) = \mathbf{U}(\boldsymbol{\alpha}, t) \tag{9}$$

$\mathbf{X}(\boldsymbol{\alpha}, t)$ is the Eulerian coordinate of the immersed membrane at time t whose Lagrangian coordinate is $\boldsymbol{\alpha}$. The immersed boundary velocity $\mathbf{U}(\boldsymbol{\alpha}, t)$ is interpolated from the fluid velocity $\mathbf{u}(\mathbf{x}, t)$ by using the same δ-function to apply the boundary force to the fluid. Equation (10) describes the immersed boundary velocity.

$$\mathbf{U}(\boldsymbol{\alpha}, t) = \int \mathbf{u}(\mathbf{x}, t) \delta(\mathbf{x} - \mathbf{X}(\boldsymbol{\alpha}, t)) \, d\mathbf{x} \tag{10}$$

2.2 Discretization

Equations 1–9 in the previous section are discretized on a uniform fixed Eulerian lattice for the fluid with a mesh width of h (the number of grid nodes is N_x, N_y and N_z in the x, y and z directions, respectively). There is also a moving Lagrangian grid for the immersed boundary with an initial mesh width $\Delta\alpha_1 = \Delta\alpha_2 = h/2$.

The D3Q19 model seen in Fig. 1 is used to discretize the BGK equation. In this model, particles can enter and exit each lattice node along eighteen different directions. The nineteenth direction represents the particles remaining at rest at the node. The particle velocity space $\boldsymbol{\xi}$ is then discretized by a set of 19 velocities (see Fig. 1).

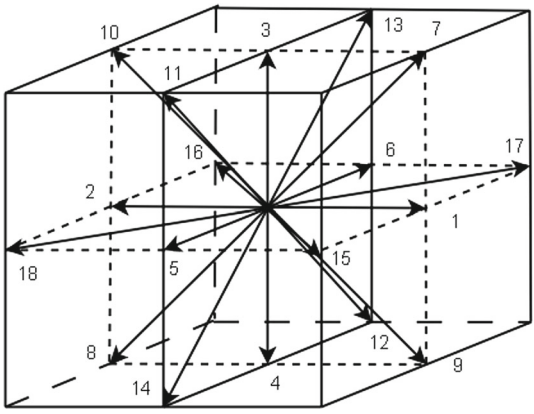


Fig. 1 D3Q19 model

Let $g_j(\mathbf{x}, t)$ be the distribution function along $\boldsymbol{\xi}_j$. A second-order space and time discretization, Eq. (11), in a Lagrangian coordinate system is applied to derive the lattice Boltzmann equation that advances $g_j(\mathbf{x}, t)$ forward by one step.

$$g_j(\mathbf{x}+\boldsymbol{\xi}_j, t+1) = g_j(\mathbf{x}, t) - \frac{1}{\tau}\left[g_j(\mathbf{x}, t) - g_j^0(\mathbf{x}, t)\right] + \left[1 - \frac{1}{2\tau}\right] w_j \left[\frac{\boldsymbol{\xi}_j - \mathbf{u}}{c_s^2} + \frac{\boldsymbol{\xi}_j \cdot \mathbf{u}}{c_s^4}\boldsymbol{\xi}_j\right] \cdot \mathbf{f}_{ib} \tag{11}$$

Here w_j is a weight for direction $\boldsymbol{\xi}_j$.

The constant $c_s = 1/\sqrt{3}$ is the speed of sound for the model. The relaxation time τ is related to the dimensionless fluid viscosity ν by the equation $\nu = \frac{2\tau-1}{6}$. The fluid velocity $\mathbf{u}$ and the force $\mathbf{f}_{ib}$ are evaluated at time t.

The density $\rho(\mathbf{x}, t)$ and momentum $(\rho\mathbf{u})(\mathbf{x}, t)$ are related to $g_j(\mathbf{x}, t)$ at each node by

$$\rho(\mathbf{x}, t) = \sum_j g_j(\mathbf{x}, t), \tag{12}$$

$$(\rho\mathbf{u})(\mathbf{x}, t) = \sum_j \xi_j g_j(\mathbf{x}, t) + \frac{\mathbf{f}_{ib}(\mathbf{x}, t)}{2}, \tag{13}$$

and the equilibrium distribution function g_j^0 is given by

$$g_j^0(\mathbf{x}, t) = \rho(\mathbf{x}, t) w_j \left[1 + 3\boldsymbol{\xi}_j \cdot \mathbf{u}(\mathbf{x}, t) + \frac{9}{2}(\boldsymbol{\xi}_j \cdot \mathbf{u}(\mathbf{x}, t))^2 - \frac{3}{2}\mathbf{u}(\mathbf{x}, t) \cdot \mathbf{u}(\mathbf{x}, t)\right]. \tag{14}$$

Assume the duration of the time step is set to 1. Let n be the time step index so that: $g^n = g(\mathbf{x}, \boldsymbol{\xi}, n)$, $\mathbf{X}^n(\boldsymbol{\alpha}) = \mathbf{X}(\boldsymbol{\alpha}, n)$, $\mathbf{u}^n = \mathbf{u}(\mathbf{x}, n)$, $p^n = p(\mathbf{x}, n)$ and $\rho^n = \rho(\mathbf{x}, n)$.

Let the flexible membrane be represented by a discrete collection of fibers whose Lagrangian coordinate is α_2. Let $\alpha_2 = q\Delta\alpha_2$, where q is an integer.

Now let each fiber be represented by a discrete collection of points whose Lagrangian coordinate is α_1. Let $\alpha_1 = m\Delta\alpha_1$, where m is an integer. The "half integer" points are given by $\alpha_1 = (m + 1/2)\Delta\alpha_1$. For any function $\phi(\alpha)$, define the operator $D_\alpha\phi$ to be the centered difference operator with respect to α.

The stretching energy and corresponding force are discretized as,

$$\mathcal{E}_s = \frac{1}{2}K_s \sum_m (|D_{\alpha_1}\mathbf{X}| - 1)^2 \Delta\alpha_1 \tag{15}$$

$$= \frac{1}{2}K_s \sum_{m=1}^{n_f-1} \left(\frac{|\mathbf{X}_{m+1} - \mathbf{X}_m|}{\Delta\alpha_1} - 1\right)^2 \Delta\alpha_1 \tag{16}$$

and

$$(\mathbf{F}_s)_l = \frac{K_s}{\Delta\alpha_1^2} \sum_{m=1}^{n_f-1} (|\mathbf{X}_{m+1} - \mathbf{X}_m| - \Delta\alpha_1) \frac{\mathbf{X}_{m+1} - \mathbf{X}_m}{|\mathbf{X}_{m+1} - \mathbf{X}_m|} (\delta_{m,l} - \delta_{m+1,l}). \tag{17}$$

Here $(\mathbf{F}_s)_l, l = 1, 2 \ldots, n_f$ is the Lagrangian force density $\mathbf{F}_s$ associated with node l. In a similar manner, the bending energy and corresponding force are discretized as,

$$\mathcal{E}_b = \frac{1}{2} K_b \sum_m |D_{\alpha_1} D_{\alpha_1} \mathbf{X}|^2 \Delta\alpha_1 \tag{18}$$

$$= \frac{1}{2} K_b \sum_{m=2}^{n_f - 1} \left[\frac{|\mathbf{X}_{m+1} + \mathbf{X}_{m-1} - 2\mathbf{X}_m|^2}{(\Delta\alpha_1)^4} \right] \Delta\alpha_1 \tag{19}$$

and

$$(\mathbf{F}_b)_l = \frac{K_b}{\Delta\alpha_1^3} \sum_{m=2}^{n_f - 1} (\mathbf{X}_{m+1} + \mathbf{X}_{m-1} - 2\mathbf{X}_m)(2\delta_{m,l} - \delta_{m+1,l} - \delta_{m-1,l}). \tag{20}$$

Here $(\mathbf{F}_b)_l, l = 1, 2 \ldots, n_f$ is the Lagrangian force density $\mathbf{F}_b$ associated with node l. n_f is the total number of grid points on the flexible membrane, and $\delta_{k,l}$ is the Kronecker symbol.

The total Lagrangian force density is $\mathbf{F}(\boldsymbol{\alpha}, t) = \mathbf{F}_s(\boldsymbol{\alpha}, t) + \mathbf{F}_b(\boldsymbol{\alpha}, t)$. The two integral relations for Eqs. (5) and (10) can now be discretized as

$$\mathbf{f}_{ib}^n(\mathbf{x}) = \sum_{\boldsymbol{\alpha}} \mathbf{F}^n(\boldsymbol{\alpha}) \delta_h(\mathbf{x} - \mathbf{X}^n(\boldsymbol{\alpha})) \, \Delta\boldsymbol{\alpha} \tag{21}$$

and

$$\mathbf{U}^{n+1}(\boldsymbol{\alpha}) = \sum_{\mathbf{x}} \mathbf{u}^{n+1}(\mathbf{x}) \delta_h(\mathbf{x} - \mathbf{X}^n(\boldsymbol{\alpha})) \, h^3. \tag{22}$$

Here the notation $\sum_{\boldsymbol{\alpha}}$ means that the sum with respect to $\boldsymbol{\alpha}$ is taken over all of the discrete collection of points. Similarly, $\sum_{\mathbf{x}}$ means that the sum with respect to $\mathbf{x}$ is taken over all discrete points of the form $\mathbf{x} = (ih, jh, kh)$. δ_h is an approximation of the Dirac δ-function. In the IB method, δ_h has the form,

$$\delta_h(\mathbf{x}) = h^{-3} \psi\left(\frac{x}{h}\right) \psi\left(\frac{y}{h}\right) \psi\left(\frac{z}{h}\right) \tag{23}$$

where h is the mesh spacing, $\mathbf{x} = (x, y, z)$.

See [4] for details regarding the choice of $\psi(r)$. With $\mathbf{U}^{n+1}(\boldsymbol{\alpha})$ known from equation (10), the flexible membrane motion equation is

$$\frac{\mathbf{X}^{n+1}(\boldsymbol{\alpha}) - \mathbf{X}^n(\boldsymbol{\alpha})}{\Delta t} = \mathbf{U}^{n+1}(\boldsymbol{\alpha}) \tag{24}$$

or

$$\mathbf{X}^{n+1}(\boldsymbol{\alpha}) = \mathbf{X}^n(\boldsymbol{\alpha}) + \mathbf{U}^{n+1}(\boldsymbol{\alpha}) \cdot \Delta t. \tag{25}$$

3 Software design

In order to understand the benefits of using GPUs, we also implement the LB-IB method using OpenMP for comparison. Since the application is written in C++, some

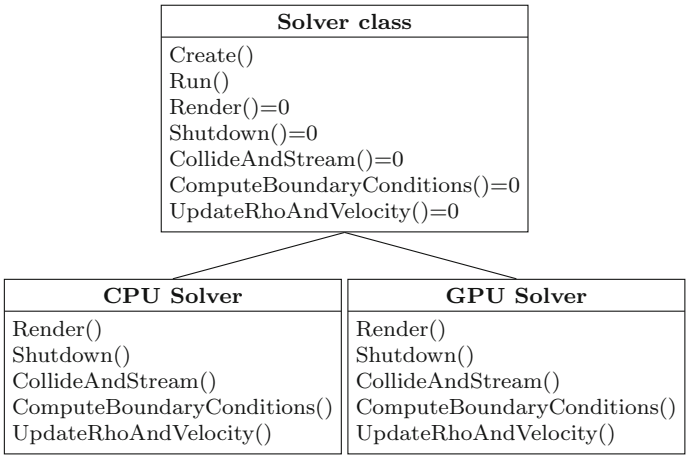


Fig. 2 Class diagram for the solver relationships

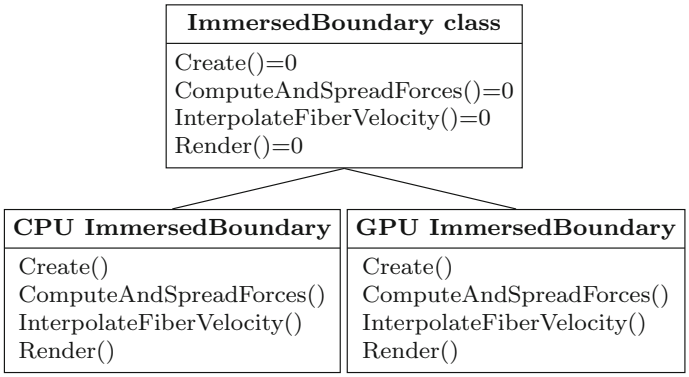


Fig. 3 Class diagram for the immersed boundary relationships

classes were designed to help with code reuse between the GPU and CPU implementations. Two major classes help the software to identify the natural hardware-specific implementation points. Figures 2 and 3 convey the essence of those two classes. The Solver abstract base class in Fig. 2 shows seven primary methods, five of which are abstract, that must be implemented to solve the lattice Boltzmann equation (the remaining two dozen methods are helper functions that are omitted from the class for clarity). A similar approach is used for the ImmersedBoundary abstract base class in Fig. 3.

The overall program structure follows in Listing 1. On lines 1 and 2, the Solver and ImmersedBoundary objects are first given an opportunity to initialize and create necessary data structures. The subclassing of the Solver allows the GPU to allocate the two large 4-dimensional matrices that store the node density values, $g(N_x, N_y, N_z, j)$, as well as the velocity and density fields, in RAM that is local to the GPU. In a similar fashion, the CPU solver allocates its matrices in host RAM. From that point forward, the Solver and ImmersedBoundary objects pass pointers to their large data structures when they need to be operated on.

```
1  Solver.Create()
2  IB.Create()
3
4  While Not Done
5  {
6     Solver.Run()-Compute field at time Tn
7        1. IB.Compute & Spread Forces              3%
8        2. Solver.Stream & Collide                61%
9        3. Solver.Compute Boundary Conditions      2%
10       4. Solver.Update Fluid Density & Velocity 16%
11       5. IB.Interpolate Fiber Velocity           2%
12       6. Solver.Compute Inlet & Outlet bc's     11%
13
14    Solver.Render()-Visualize field at time Tn    5%
15 }
```

Listing 1: Pseudo code for main program logic

After this initialization, the Run() method computes the field at time T_n on the Tesla card (line 6), while the Render() method displays and visualizes the generated field (line 14). The 6 steps associated with computing the field in lines 7–12 are implemented with the subclassed form of the IB (Immersed Boundary) class or the Solver (lattice Boltzmann) class.

The use of the abstract base class makes it easier to reason about the two different hardware platforms. Common variables like the dimensions of the domain, the Reynold number, or the current time step are naturally allocated in the base classes, while hardware-specific details are in the subclasses. In fact, this class design would make it fairly straightforward to evaluate the performance of the LB-IB algorithm on a distinct accelerator such as the Xeon Phi from Intel.

The values on the right of each line represent the execution duration as a percentage of the total time needed to generate one frame for a domain size of 256^3. Our test case deals with the deformation of a flexible mesh that is composed of an orthogonal collection of fibers. Listing 1 suggests that the streaming and collision step in the lattice Boltzmann method is the most expensive function to implement. For our problem, the computation of the fluid forces and the movement of the fibers required just 3% of *the frame duration*. The frame duration is the amount of time required for the computing GPU to advance the LB-IB algorithm, plus the amount of time required for the visualization GPU render the three-dimensional scene. We expect more complex immersed objects will consume more of the computing time. For example, in an extreme case, if the number of nodes in the immersible structure is comparable to the number of fluid grid nodes, the computation of the interactive forces may then become comparable to or even dominate the execution time.

4 Implementation of LB-IB and visualization on CPU and GPU

4.1 OpenMP details on CPU

Given the challenges associated with writing and debugging GPU code, we decided to implement the OpenMP version first. This approach helped us identify and solve

race conditions and performance issues before we moved to the GPU. It also helped us recognize the two base classes in Figs. 2 and 3.

The primary difference between the two implementations lies in the data parallel approach used on the GPU. On the GPU, we typically launched one thread for each node in the 3D domain that required computing. The equivalent implementation with OpenMP on CPU requires 3 nested for-loops. However, the outermost loop (for the Z axis) is preceded by the "#pragma omp parallel" directive to request distribution of the work across multiple CPU cores.

Many of the lessons learned while implementing the OpenMP code could be applied to the GPU code. For example, the GPU code, like the OpenMP code, merged the collision and streaming steps into a single step, thereby eliminating a large block of memory reads and writes. In addition, the AOS (array of structures) implementation of the fluid velocity field that negatively impacted the OpenMP code also affected the GPU code. The cache friendly solution was to switch to an SOA (structure of arrays) approach which minimized cache line reloads on the CPU. On the GPU, the SOA approach effectively reduced the number of memory transactions by a factor of 3. For example, when a warp of 32 cores generated 32 memory addresses to access the X component of the fluid velocity, the SOA approach guaranteed that those 256 bytes were physically contiguous in RAM. In contrast, the AOS approach distributed those same 32 memory addresses across 768 bytes because the stride between logically adjacent X components was now 24 instead of 8.

Although C++ natively supports three-dimensional arrays, we wrapped our arrays in a class so that subscript checking could be enabled in the debug build of the application. This greatly simplified the search for errant logic that occasionally indexed before the beginning or past the end of an array. Given that we consciously accessed the domain in *Z*-*Y*-*X* axis order, we could then insure that accesses to sequential nodes were physically contiguous in memory which again further minimized cache line reloads on the CPU.

4.2 GPU details

The GPU used for implementing the LB-IB algorithms and computing the 3D vector field was the Tesla K40c. This card features 2880 CUDA cores, 1.4 TFLOPS double precision (DP), and 12 GB of RAM. The development environment consisted of CUDA 7.5 and Visual Studio 2013 under Windows 7/64.

Computation of the bending and stretching forces (that accompany the spreading of the forces) in line 7 of Listing 1 was a straight forward implementation of Eqs. (17) and (20). A data parallel approach was taken where one GPU thread is dedicated to each point on the Lagrangian grid with synchronization primitives inserted as appropriate.

The subsequent spreading of the forces from the fibers to the fluid in Eq. (21) is described in kernel Listing 2. First, notice in lines 20–25 how all of the accesses to global memory on the GPU are initiated as early as possible in the kernel. This aids in filling the memory controller pipeline and minimizes stalls later in the code due to unavailable operands.

Next, from the host computer's perspective, we simply launch as many GPU threads as there are fibers and points per fiber. Each thread will then be responsible for accessing the points from the Eulerian grid and accumulating them from the neighboring fiber nodes on the Lagrangian grid. The potential problem with this approach is that a race condition can occur. Fortunately, modern GPUs have an atomic add instruction that makes this accumulation indivisible as shown on lines 58–60 of Listing 2.

Finally, Eq. (21) initially concerned us from a performance standpoint because the Eulerian force $\mathbf{f_{ib}}$ is computed by way of a smoothing function $\delta_h(\mathbf{x})$ that accesses the $4 \times 4 \times 4$ cube of values around each node of the Lagrangian grid. With $\delta_h(\mathbf{x})$ defined by Eq. (23), it becomes apparent that the non-coalesced nature of the 64 coefficients may negatively impact the function. However, with line 7 of Listing 1 consuming 3% of the execution time, we deferred further analysis until the number of Lagrangian grid points significantly increased.

The lattice Boltzmann method consists of a collision step, a streaming step, and a boundary computation step. Our first implementation on the GPU followed this sequence. However, the latency associated with reading or writing GPU memory can take between 400 and 600 clock cycles. As such, it is in our interest to minimize redundant reads and writes. In our second GPU implementation, we merged the collision and streaming steps into a single step, thereby eliminating the extra reading and writing of $8 \times 19 \times 256^3$ bytes. This resulted in a *twofold increase* in the execution speed of the method, which further reinforced our belief that the LB method is memory bound and not arithmetic bound.

Line 10 of Listing 1 shows that updating the fluid density and velocity consumed about 16% of the GPU time for one frame. Equations (12) and (13) describe the process. Implementing the two equations directly would result in inefficient GPU code. The GPU will block only when an operand is not available. The better solution is to read the 19 distribution values into an array to fill the memory controller's read pipeline. The final CUDA kernel is described in Listing 3. Lines 28–33 focus on performing all memory accesses. Lines 40–51 focus on the actual computation which are not likely to stall because the memory-based operands should now be in registers.

Line 11 of the pseudocode in Listing 1 describes the interpolation of the fiber velocity in terms of Eq. (22). When combined with Eq. (25), the fibers are repositioned to their new points in space. Equation (22), like Eq. (21), also concerned us from a performance standpoint because the Lagrangian velocity field $\mathbf{U}$ is computed by way of $\delta_h(\mathbf{x})$ that accesses the $4 \times 4 \times 4$ cube of values around each node of the Eulerian grid. The access pattern is nearly identical to that used in lines 41–63 of the SpreadForcesKernel in Listing 2. However, with line 11 of Listing 1 consuming 2% of the execution time, we again deferred optimization until the number of Lagrangian grid points significantly increased.

4.3 Visualization details

In our software implementation, the Tesla GPU card generates the velocity and pressure fields at time T_n and then the Quadro GPU card uses custom GLSL shaders [60] to generate the display. The code was explicitly designed to separate the two functions.

```
__global__
void SpreadForcesKernel(

  double *fx, double *fy, double *fz,
  double *filxn1, double *filyn1, double *filzn1,
  double *ffx, double *ffy, double *ffz,
  double dx, double dy,double dz,
  int fffwidth,   // width of ff? 3D matrices
  int fffheight,  // height of ff? 3D matrices
  int NumberOfFibers,
  int PointsPerFiber)
{
const int kf = blockIdx.x*blockDim.x + threadIdx.x;
const int jf = blockIdx.y*blockDim.y + threadIdx.y;

if (jf >= NumberOfFibers) return;
if (kf >= PointsPerFiber) return;

const int kfjfinx    = kf + jf*PointsPerFiber;
const double filxn1t = filxn1[kfjfinx];
const double filyn1t = filyn1[kfjfinx];
const double filzn1t = filzn1[kfjfinx];
const double fxt     = fx[kfjfinx];
const double fyt     = fy[kfjfinx];
const double fzt     = fz[kfjfinx];

const double cx = 3.1415926535  / (2.0 * dx);
const double cy = 3.1415926535  / (2.0 * dy);
const double cz = 3.1415926535  / (2.0 * dz);
const double cf = 1.0 / (64.0 * dx*dy*dz);
const double K0 = cf;

const int istart = static_cast<int>(floor(filxn1t/dx-2)+1);
const int jstart = static_cast<int>(floor(filyn1t/dy-2)+1);
const int kstart = static_cast<int>(floor(filzn1t/dz-2)+1);

const int istop  = istart + 4;
const int jstop  = jstart + 4;
const int kstop  = kstart + 4;

for (int i = istart; i < istop; i++)
{
  const double rx = dx*static_cast<double>(i)-filxn1t;
  const double K1 = (1.0 + COS(cx*rx)) * K0;

  for (int j = jstart; j < jstop; j++)
  {
    const double ry   = dy*static_cast<double>(j)-filyn1t;
    const double K1K2 = K1*(1.0 + COS(cy*ry));

    for (int k = kstart; k < kstop; k++)
    {
      const double rz = dz*static_cast<double>(k)-filzn1t;
      const double K3 = ((REAL)1.0 + COS(cz*rz))*K1K2;

      const int inx  = i + j*fffwidth + k*fffwidth*fffheight;

      atomicAdd(ffx+inx, fxt*K3);
      atomicAdd(ffy+inx, fyt*K3);
      atomicAdd(ffz+inx, fzt*K3);
    }
  }
}
}
```

Listing 2: CUDA kernel for spreading forces

```
__global__
void UpdateRhoAndVelocityKernel(
  double *fin,
  double *UField, double *VField, double *WField,
  double *Rho,
  double *ffx, double *ffy, double *ffz,
  double dt,
  double gl,
  int width,
  int height,
  int zdim)
{
   const int x = GetXIndex();
   const int y = GetYIndex();
   const int z = GetZIndex();

   if (x < 2 || x >= width-2) return;
   if (y < 2 || y >= height-2) return;
   if (z < 2 || z >= zdim-2) return;

   const int inx     = x + width*(y + z*height);
   const int NumCells= width*height*zdim;
   double F[19];

   for (int i = 0; i < 19; i++)
      F[i] = fin[inx + i*NumCells];

   double ffxt = ffx[inx];
   double ffyt = ffy[inx];
   double ffzt = ffz[inx];
   double SumF = 0.0;
   double SumX = 0.0;
   double SumY = 0.0;
   double SumZ = 0.0;

   for (int i = 0; i < 19; i++)
   {
     double Q;
     SumF += (Q = F[i]);
     SumX += Xi[i].x*Q;
     SumY += Xi[i].y*Q;
     SumZ += Xi[i].z*Q;
   }

   double X1   = (SumX+0.5*dt*ffxt)/SumF;
   double Y1   = (SumY+0.5*dt*ffyt)/SumF;
   double Z1   = (SumZ+0.5*dt*(ffzt+SumF*gl))/SumF;
   Rho[inx]    = SumF;
   UField[inx] = X1;
   VField[inx] = Y1;
   WField[inx] = Z1;
}
```

Listing 3: CUDA kernel for updating fluid density and velocity

From Listing 1, it seemed that with visualization taking only 5% of the frame time, the dual GPU approach probably was not necessary for this example. However, as our immersed objects become more complex and more numerous, we have an architecture in place *that will allow us to overlap the computing and visualization phases*. This will amount to the Tesla card computing the field at time T_n while the Quadro visualizes time T_{n-1}. In addition, as we anticipate to implement isosurfaces or stream ribbons to visualize vortices [61] in the future, these calculations will be performed exclusively on the Quadro card.

All of the 3D graphics were implemented using OpenGL. OpenGL is a cross-platform API for rendering 2D and 3D vector graphics. The original API is called the Direct Mode API. It is simple to use if one is familiar with 3D graphics concepts. Unfortunately, the Direct Mode API is poorly matched for modern GPU hardware. Modern hardware prefers to be handed blocks of thousands or millions of vectors at time. As a result, starting with OpenGL 3.0, the Direct Mode API was deprecated in preference for the new API. The new API is more challenging to use. However, some of our simple tests showed that for a given GPU card like a Quadro K5000, the drawing rate was minimally 10 times faster than the Direct Mode API. In this work, we chose to implement the graphics using as much of the new API as possible. As mentioned earlier, we took this implementation path in anticipation of using a large number of complex, immersed boundary objects in our future work.

We follow [62] on visualization of vector fields. The author discussed how vector glyphs can be used as trajectories of imaginary particles that are released into the vector field over a short period of time δt. This was one of the two visualization techniques that we have realized. The author then describes a broader set of tools known as *stream objects* whose purpose is to utilize those same trajectories over a longer period of time.

The other visualization technique we implemented is the *streamline* visualization. For a time-independent vector field, a streamline is a curved path starting from a given point $\mathbf{x_0}$ which is tangent to $\mathbf{v}$, the vector field. If a streamline is modeled as a parametric function $S(\tau) = \mathbf{x}(\tau)$, where τ represents the arc-length coordinate along the curve, then a streamline obeys the equation

$$\frac{d\mathbf{x}(\tau)}{d\tau} \times \mathbf{v}(\mathbf{x}(\tau)) = 0. \tag{26}$$

This can also be expressed as the following ODE,

$$\frac{d\mathbf{x}(\tau)}{d\tau} = \frac{\mathbf{v}(\mathbf{x}(\tau))}{|\mathbf{v}(\mathbf{x}(\tau))|} \tag{27}$$

with the initial condition $\mathbf{x}(s = 0) = \mathbf{x_0}$ and the constraint $s \in [0, S_{\max}]$. When Eq. (27) is integrated over τ from 0 to s, we have the equation

$$\mathbf{x}(s) = \mathbf{x}(0) + \int_0^s \frac{\mathbf{v}(\mathbf{x}(\tau))}{|\mathbf{v}(\mathbf{x}(\tau))|}\, d\tau, \tag{28}$$

with $\mathbf{x}(s = 0) = \mathbf{x_0}$.

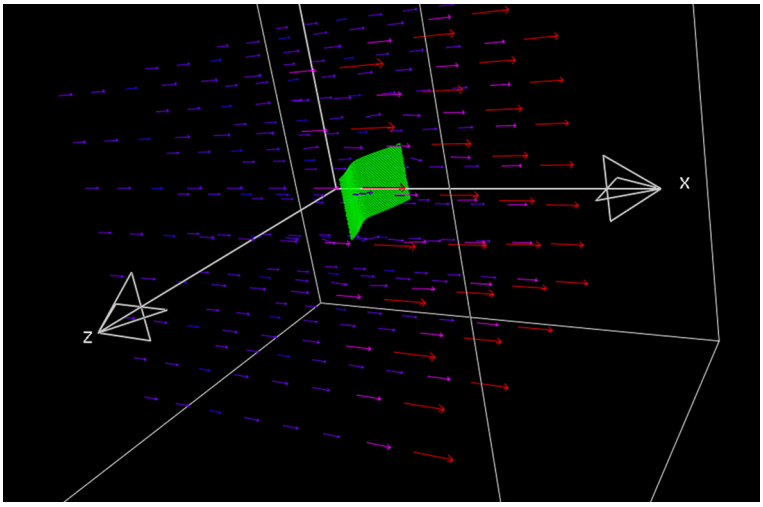


Fig. 4 User interface with glyphs

Equation (27) can be implemented using Euler's method as follows,

$$\int_0^T \frac{\mathbf{v}(\mathbf{x}(t))}{|\mathbf{v}(\mathbf{x}(t))|}\,dt \approx \sum_{i=1}^{N=T/\delta t} \frac{\mathbf{v}(x_i)}{|\mathbf{v}(x_i)|}\delta t, \quad \mathbf{x}_i = \mathbf{x}_{i-1} + \mathbf{v}_{i-1}\delta t. \tag{29}$$

However, the global error of Euler's method, $\mathcal{O}(\delta t)$, suggests we pursue a better integrator. The Runge–Kutta 2 (i.e., RK2) and the RK4 integrators have global errors of $\mathcal{O}(\delta t^2)$ and $\mathcal{O}(\delta t^4)$, respectively. Therefore, we have implemented both in the event we suspected drift in the placement of the streamlines.

Our visualization subsystem is capable of generating the 3D vector field using glyphs or streamlines [61,63–66]. The glyph-based approach draws an arrow at a point in 3D space that is oriented with the vector flow as shown in Fig. 4. This approach is easy to implement but suffers from the problem of visual clutter. Even with the ability to rotate and zoom our 3D domain during simulation, it can be difficult to discern details of the underlying field.

Listing 4 describes the algorithm. Lines 6–10 scan the velocity field and search for the longest vector that is a fluid node. Lines 13–19 then loop one more time over the velocity field to find the *head* and *tail* of each fluid node vector. After the *head* is normalized, an arrow is drawn.

Streamlines, on the other hand, show where the vector flow has come from and where it is going to. By changing the length of the streamlines during the computation, we can more easily accentuate features such as vortices. The streamlines are always tangent to the vector field, and fluid never crosses a streamline. Figure 5 shows a screen shot of the user interface while the simulation is running and the field is viewed with streamlines.

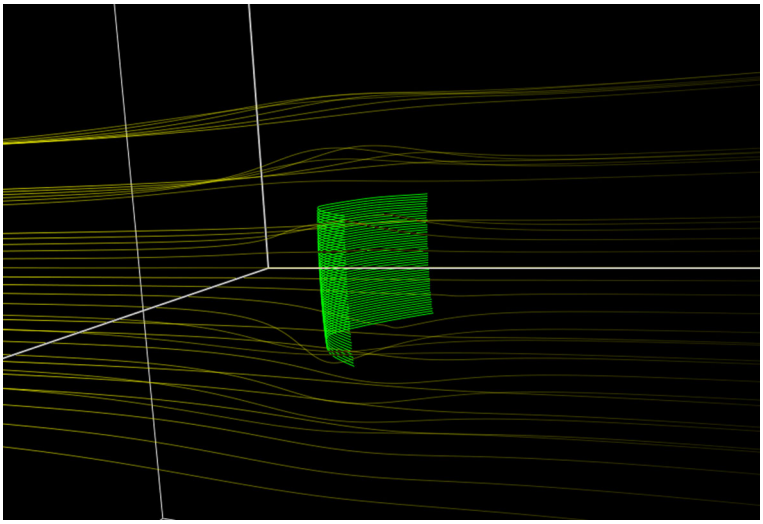


Fig. 5 User interface with streamlines

```
DrawGlyphs(UData, VData, WData, IsBoundaryNode, Width, Height, ZDim)
{
  MaxLen = 0

  // Find the length of the largest vector
  for Z = 1 to ZDim
    for Y = 1 to Height
      for X = 1 to Width
        if IsBoundaryNode(X,Y,Z) = 0
          MaxLen = Max(MaxLen, VectorLength(X,Y,Z))

  // Draw the vectors properly scaled
  for Z = 1 to ZDim
    for Y = 1 to Height
      for X = 1 to Width
        if IsBoundaryNode(X,Y,Z) = 0
          Head = [UData(X,Y,Z),VData(X,Y,Z),WData(X,Y,Z)] / MaxLen
          Tail = X,Y,Z
          DrawArrow(Head,Tail)
}
```

Listing 4: Pseudo code to draw vector glyphs

Listing 5 describes the streamline algorithm. During the initialization of the application, a number of 3D points called *seeds* are calculated. These seeds represent the starting point for the stream lines. The seeds are uniformly distributed across the inlet plane of the 3D domain. When the *DrawStreamLines* procedure is called, the logic selects a seed and then performs an RK2 integration over $PathLineLength$ points. This essentially implements Eq. (28). When the *TraceRK2* procedure returns, the *StreamLine* variable contains a collection 3D points that trace the path. Line 6 then calls the *Draw* procedure to display the path.

Table 1 Simulation parameters for Figs. 9, 10, 11 and 12

Time step	N_x	N_y	N_z	n_f	Initial velocity	Re	K_s	K_b
100,000	256	256	256	104	0.03	150	0.0004	0.0005
100,000	256	256	256	104	0.03	150	0.0004	0.0015
100,000	256	256	256	104	0.03	150	0.0004	0.0032
100,000	256	256	256	104	0.03	150	0.0004	0.0050

```
1  DrawStreamLines(}
2  {
3    for i = 1 to NumberOfSeeds
4      aSeed = Seeds[i]
5      TraceRK2(aSeed, StreamLine)
6      Draw(StreamLine)
7  }
8
9  TraceRK2(aSeed, StreamLine)
10 {
11   PathLineLength = 0
12   CurrentPoint   = aSeed
13
14   for i=0, PathLineLength<MaxPathLineLength, i++
15      bool InSide = InterpolateVelocity(CurrentPoint, vA)
16      if not InSide return
17
18      StreamLine.add(CurrentPoint)
19
20      if Length(vA) < 1E-5 return
21
22      vecB = CurrentPoint + vA*StepSize
23      bool InSide = InterpolateVelocity(vecB, vB)
24
25      if Length(vB) < 1E-5 return
26
27      v = (vB + vA) * 0.5
28
29      CurrentPoint   = CurrentPoint + v*StepSize
30      PathLineLength = PathLineLength + StepSize
31 }
```

Listing 5: Pseudo code to draw stream lines

The *TraceRK2* procedure performs the actual integration. The procedure keeps track of the *PathLineLength*. Line 14 traverses the vector field until the *PathLineLength* becomes too long. Otherwise line 15 calls *InterpolateVelocity* which performs trilinear interpolation of the vector field at the current point. The interpolated value is appended to the *StreamLine* on line 18. Line 20 checks the length of the interpolated velocity and exits the procedure if the magnitude is smaller than 10^{-5}. The algorithm updates the *CurrentPoint* which advances it on the field. Finally, the *PathLineLength* is updated by the step size.

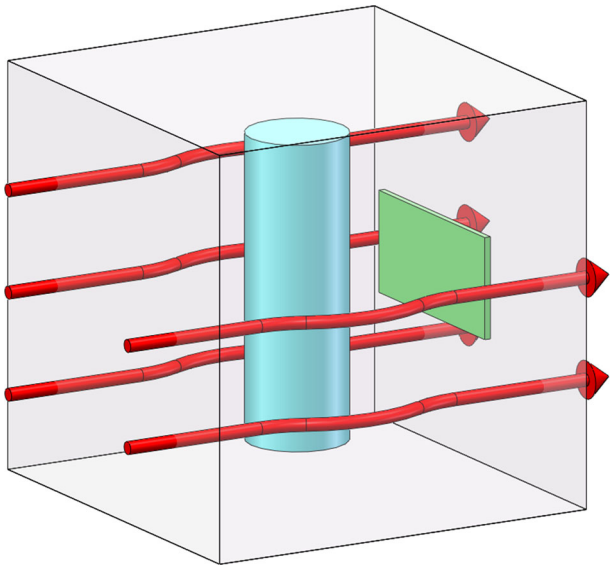


Fig. 6 Cartoon of the 3D field flowing around a cylinder and interacting with a flexible mesh

This visualization technique has several useful properties. First, it is easy to implement which makes it amenable to implementation on the host CPU or the graphics GPU. Second, if the hardware permits, all of the stream lines can be computed in parallel.

As mentioned earlier, the placement and density of the glyphs and stream lines can be changed while the simulation is running. In addition, one key stroke will pause the simulation, while another key stroke will toggle between the two visualization techniques. This type of functionality supports the exploratory nature of the application.

5 Results

The results of our real-time GPU implementation of the 3D LB-IB method are illustrated by the example problem shown in Fig. 6: a 3D viscous incompressible fluid flows around a circular rigid cylinder with a tethered flexible mesh placed behind. The fluid enters from the left, flows around a cylinder, interacts with the flexible mesh, and then exits on the right. The remaining four faces of the domain implement a no-slip boundary condition. The flexible mesh is tethered in space along a vertical line that divides the mesh in half. The flexible mesh consists of 52 fibers oriented vertically, 52 fibers oriented horizontally and 103 points per fiber. The simulation shows how

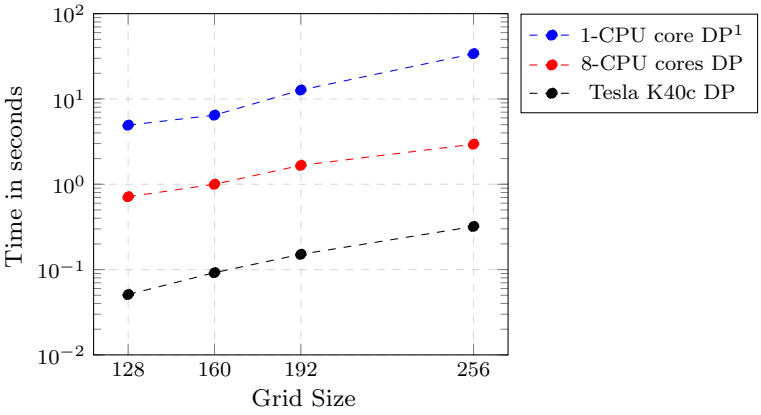


Fig. 7 Average time step execution times for 104 fibers and 103 points/fiber

the mesh folds in half under interaction with the flow field and the complicated flow patterns behind the mesh.[1]

To measure the performance, we started with 3 configurations (a single-CPU core, 8-CPU cores under OpenMP, and a single GPU). Each configuration was executed for our example problem in Fig. 6 with a grid size of 128, 160, 192 and 256. Each test ran for 100 time steps. We then computed the average duration per time step by dividing the execution time by 100.

The execution times for the double-precision implementations are in Fig. 7. For a given grid size, the GPU is approximately 10 times faster than 8 cores using OpenMP, and the 8 core OpenMP version is approximately 8 times faster than a single core. As shown earlier, the rendering performed by the Quadro GPU consumed only 5% of the total compute duration for each frame. In this example, we explored how the flow around a cylinder affects the flexible mesh. However, our framework can be adapted to explore almost all FSI problems involving a viscous incompressible fluid and an immersed elastic structure, including the flow around a sphere, a cube or a torus.

The simulator has numerous parameters that control the behavior of the LB-IB method. These parameters include the domain dimensions N_x, N_y, N_z, the number of fibers n_f, initial fluid field velocity, the Reynolds number Re, fiber stretching coefficient K_s and the fiber bending rigidity K_b. All of these parameters can be modified for experimentation purposes or for parametric studies. Table 1 shows the parameters for the four simulations we performed.

Figures 9, 10, 11 and 12 demonstrate the mesh final position, the mesh shape, and the flow field (by streamlines) at time 100,000 (in lattice Boltzmann units). These figures reveal that with a larger bending coefficient the mesh is deformed less at the final equilibrium state, and the flow patterns behind the mesh become more and more chaotic and complicated. To highlight the interactive features of our software, Figs. 13, 14, 15 and 16 display the time evolution of mesh deformation and the flow field with some of the user interface elements present. The bounding box is drawn to

[1] Lenovo D30, 8 core E5-2609@2.4GHz, 32GB RAM, Windows 7/64.

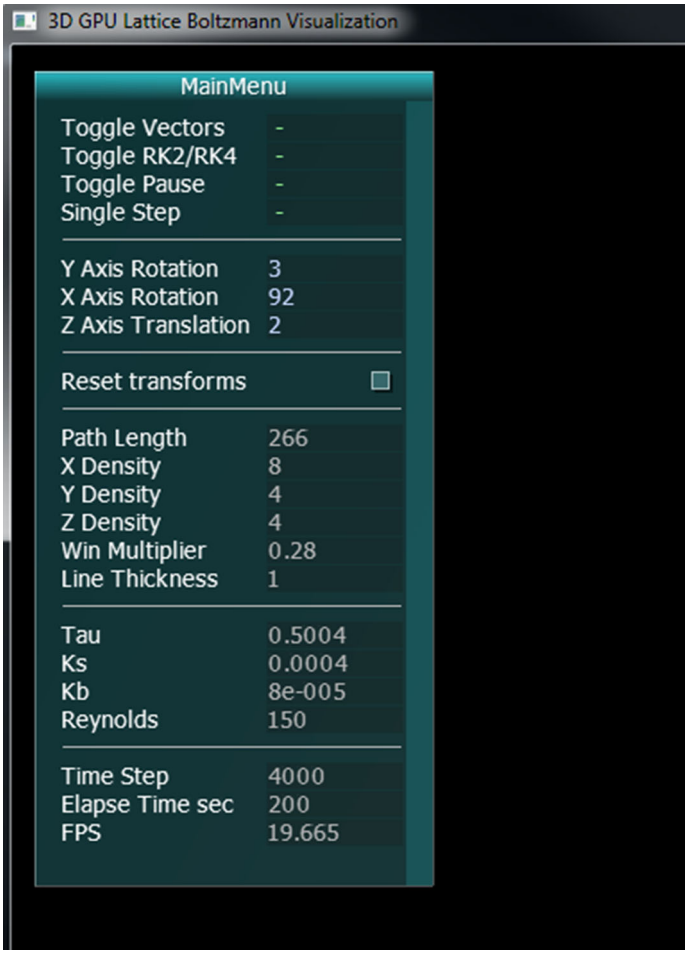


Fig. 8 The menu in the interface

help the user understand the extent of the domain. Arrows for the X, Y and Z axis are drawn to better understand the orientation of the simulation. Finally, a menu of options (Fig. 8) is presented (along with the current state of the simulation) so that the user can interactively change

- the vector glyphs or the stream lines
- the use of an RK2 versus RK4 stream line integrator
- the ability to pause the simulation on a time step
- the ability to single step the simulation.

The view point of the user is changed by simply dragging the mouse in the X and Y direction so that the bounding box is rotated. The simulation parameters are shown in Table 2.

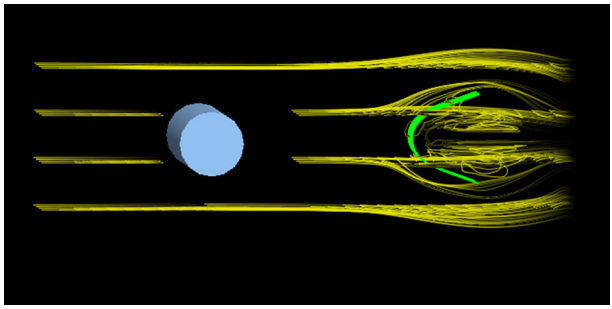


Fig. 9 $K_b = 0.0005$ at time step 100,000

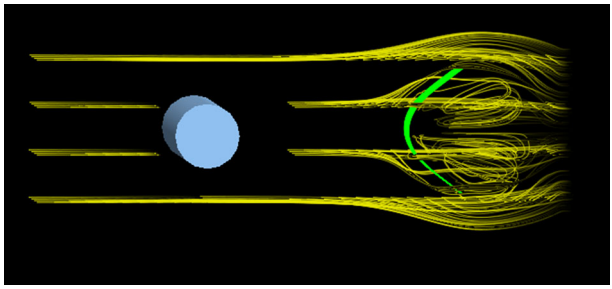


Fig. 10 $K_b = 0.0015$ at time step 100,000

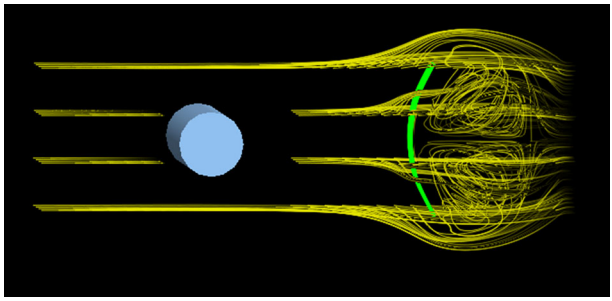


Fig. 11 $K_b = 0.0032$ at time step 100,000

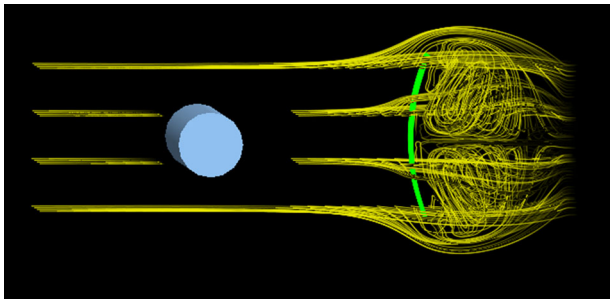


Fig. 12 $K_b = 0.0050$ at time step 100,000

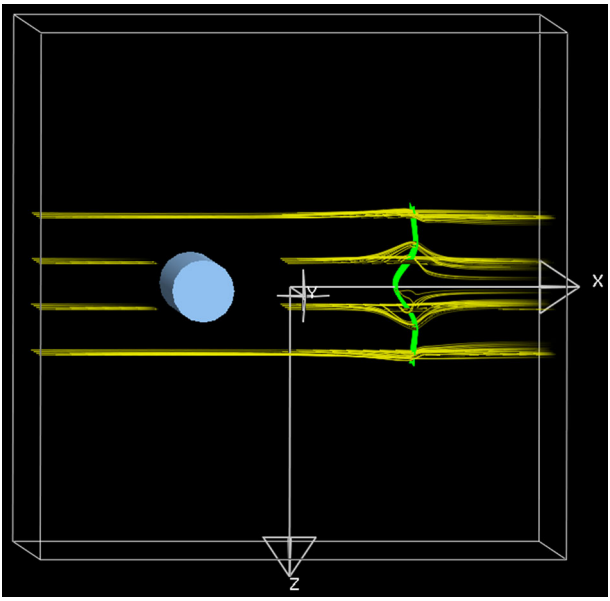


Fig. 13 $K_b = 0.0005$ at time step 4000

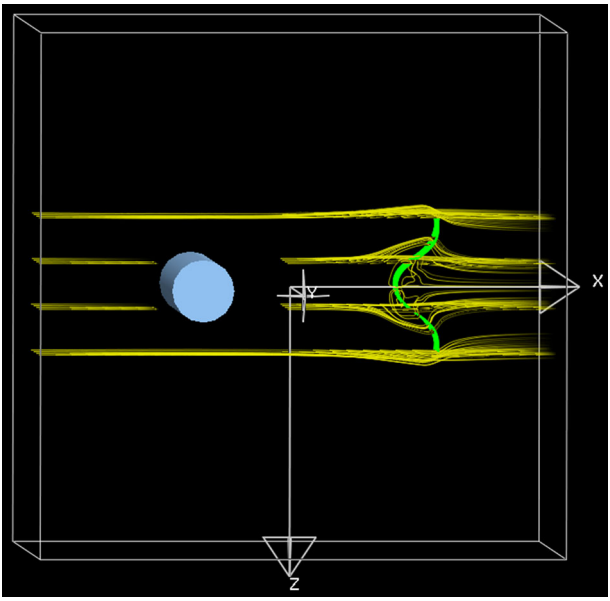


Fig. 14 $K_b = 0.0005$ at time step 8000

Validation of numerical codes like LB-IB can be performed through the use of convergence checking. In this approach, a series of gradually refined grids ensures that the results are reliable and accurate. In our case, we needed to prove that our

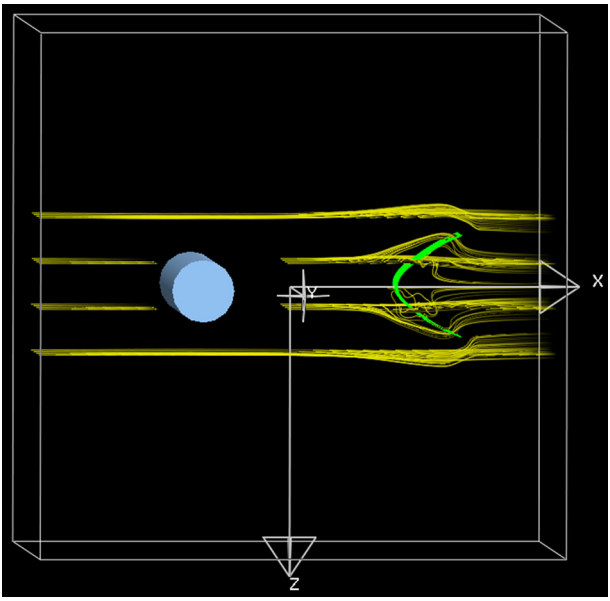


Fig. 15 $K_b = 0.0005$ at time step 12,000

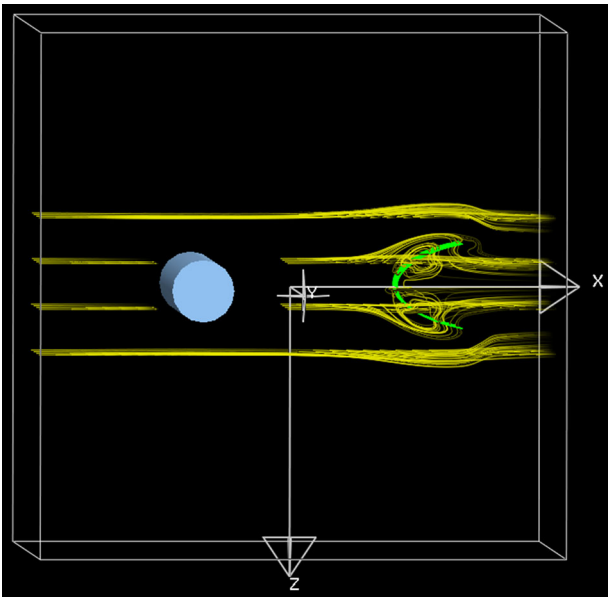


Fig. 16 $K_b = 0.0005$ at time step 16,000

solutions were valid *and* that the CPU and GPU versions were sufficiently identical. Given that floating point arithmetic is not associative, it can be challenging to produce CPU results that match the GPU. Since we had access to the code produced by Zhu

Table 2 Simulation parameters for Figs. 13, 14, 15 and 16

Timestep	N_x	N_y	N_z	n_f	Initial velocity	Re	K_s	K_b
4000	256	256	256	104	0.03	150	0.0004	0.00008
8000	256	256	256	104	0.03	150	0.0004	0.00008
12,000	256	256	256	104	0.03	150	0.0004	0.00008
16,000	256	256	256	104	0.03	150	0.0004	0.00008

[31] (that had been previously validated with convergence checking), we constantly compared our results with theirs.

Our first step was to develop our 1-core OpenMP code and configure it with parameters identical to [31]. Then, as each new version of the code was produced, we would compare all 507,904 values in our $124 \times 64 \times 64$ test domain with that of Zhu's at time step 1, 100, and 1000. Our criteria were that the results needed to match to three decimal places (the codes were implemented using double-precision floating point). Since our OpenMP implementation was carefully thought through, our 1-core, 4−core and 8-core versions were identical. This was relatively easy to achieve because the code accessed and operated on its parameters in the same order. In fact, when the three versions did not match, it was usually due to a logic error or race condition.

With these three versions in place, we then implemented the GPU version. Again, with each new version of the GPU code, we would compare the values from our $124 \times 64 \times 64$ domain to those produced by the OpenMP version. We again used a criteria that our results needed to match to three decimal places.

6 Summary and future work

This paper presents a dual GPU interactive implementation of the lattice-Boltzmann-based immersed boundary method for fluid–structure interaction problems in three dimensions. The implementation is demonstrated on a 3D, viscous, incompressible flow past a deformable mesh behind a rigid cylinder of circular section. Our software works for many FSI problems involving an incompressible viscous fluid and an elastic structure (rigid or deformable). To the best of our knowledge, our software is the first GPU tool integrating visualization and computing in CFD. It is expected to shorten time to solution and speed up scientific discoveries in the FSI field.

Our single-CPU core, 8-CPU core and Tesla implementations are compared from a performance perspective using double-precision arithmetic. Our simulations demonstrate an 80-fold improvement of the single GPU over the single-CPU core. Our base class design for the LB-IB algorithm greatly simplifies the hardware-specific implementations. A single application was created that addressed both hardware platforms for the purpose of performance monitoring and comparison.

Our GPU implementation of the smoothed Dirac delta function for identifying influence and dependent domains of a Lagrangian point on the structure (for force

spreading and velocity interpolation) may need to be optimized when the immersed body becomes so complex that the number of Lagrangian structure grid points are comparable to the Eulerian fluid grid points. Otherwise the software performance may deteriorate. This is a nice future work.

Our dual GPU approach shows that there are a large number of visualization cycles available for more complex immersed objects. As another future work, we plan to explore distributing the computation across multiple GPUs, expanding the visualization capabilities to include isosurfaces of quantities such as vorticity, volume rendering on the visualization GPU, and overlapping the computation with the rendering.

Acknowledgements The authors would like to thank the NSF support under the Grant award Number DMS-1522554.

References

1. Tian FB, Luo H, Zhu L, Lu XY (2010) Interaction between a flexible filament and a downstream rigid body. Phys Rev E 82:026301
2. Espinha LC, Hoey DA, Fernandes PR, Rodrigues HC, Jacobs CR (2014) Oscillatory fluid flow influences primary cilia and microtubule mechanics. Cytoskeleton 71:435–445
3. Huang S, Li R, Li QS (2013) Numerical simulation on fluid–structure interaction of wind around super-tall building at high reynolds number conditions. Struct Eng Mech Int J 46:197–212
4. Peskin CS (2002) The immersed boundary method. Acta Numer 11:409
5. Mittal R, Iaccarino G (2005) Immersed boundary methods. Annu Rev Fluid Mech 37:239–261
6. LeVeque RJ, Li ZL (1997) Immersed interface methods for Stokes flows with elastic boundaries or surface tension. SIAM J Sci Comput 18:709–735
7. Cortez R (2000) A vortex/impulse method for immersed boundary motion in high Reynolds number flows. J Comput Phys 160:385–400
8. Wang XS (2006) From immersed boundary method to immersed continuum method. Int J Multiscale Comput Eng 4:127–145
9. Zhang L, Gersternberger A, Wang X, Liu WK (2004) Immersed finite element method. Comput Methods Appl Mech Eng 193:2051
10. Hughes TJR, Liu WK, Zimmermann TK (1981) Lagrangian–Eulerian finite element formulation for incompressible viscous flows. Comput Methods Appl Mech Eng 29:329–349
11. Glowinski R, Pan T, Periaux J (1994) A fictitious domain method for Dirichlet problem and applications. Comput Methods Appl Mech Eng 111:1994
12. Sulsky D, Chen Z, Schreyer HL (1994) A particle method for history-dependent materials. Comput Mech Appl Mech Eng 118:179–197
13. Cottet G-H, Maitre E (2006) A level set method for fluid–structure interactions with immersed surfaces. Math Models Methods Appl Sci 16:415–438
14. Kim J-D, Li Y, Li X (2013) Simulation of parachute FSI using the front tracking method. J Fluids Struct 37:100–119
15. Peskin CS (1972) Flow patterns around heart valves: a digital computer method for solving the equations of motion, vol 378. PhD thesis. Physiology, Albert Einstein College of Medicine, University of Microfilms, pp 72–30
16. Peskin CS (1977) Flow patterns around heart valves; a numerical method. J Comput Phys 25:220
17. McCracken MF, Peskin CS (1980) A vortex method for blood flow through heart valves. J Comput Phys 35:183–205
18. Rosar ME, Peskin CS (2001) Fluid flow in collapsible elastic tubes: a three-dimensional numerical model. New York J Math 7:281–302
19. Roma AM, Peskin CS, Berger MJ (1999) An adaptive version of the immersed boundary method. J Comput Phys 153:509–534
20. Lai MC, Peskin CS (2000) An immersed boundary method with formal second order accuracy and reduced numerical viscosity. J Comput Phys 160:705

21. Griffith BE, Peskin CS (2015) On the order of accuracy of the immersed boundary method: higher order convergence rates for sufficient smooth problems. J Comput Phys 208:75–105
22. Zhu L, Peskin CS (2002) Simulation of a flexible flapping filament in a flowing soap film by the immersed boundary method. J Comput Phys 179:452–468
23. Kim Y, Peskin CS (2007) Penalty immersed boundary method for an elastic boundary with mass. Phys Fluids 19:053103
24. Fauci LJ, Fogelson AL (1993) Truncated Newton methods and the modeling of complex elastic structures. Commun Pure Appl Math 46:787
25. Taira K, Colonius T (2007) The immersed boundary method: a projection approach. J Comput Phys 225:2118–2137
26. Mori Y, Peskin CS (2008) Implicit second-order immersed boundary method with boundary mass. Comput Methods Appl Mech Eng 197:2049–2067
27. Hao J, Zhu L (2010) A lattice Boltzmann based implicit immersed boundary method for fluid–structure-interaction. Comput Math Appl 59:185–193
28. Hao J, Zhu L (2011) A 3D implicit immersed boundary method with application. Theor Appl Mech Lett 1:062002
29. Lim S, Ferent A, Wang XS, Peskin CS (2008) Dynamics of a closed rod with twist and bend in fluid. SIAM J Sci Comput 31:273–302
30. Atzberger PJ, Kramer PR, Peskin CS (2006) A stochastic immersed boundary method for biological fluid dynamics at microscopic length scale. J Comput Phys 224:1255–1292
31. Zhu L, He G, Wang S, Miller L, Zhang X, You Q, Fang S (2011) An immersed boundary method based on the lattice Boltzmann approach in three dimensions with application. Comput Math Appl 61:3506–3518
32. Feng ZG, Michaelides EE (2005) Proteus: a direct forcing method in the simulations of particulate flows. J Comput Phys 202:20–51
33. Tian FB, Luo H, Zhu L, Liao JC, Lu X-T (2011) An efficient immersed boundary-lattice Boltzmann method for the hydrodynamic interaction of elastic filaments. J Comput Phys 230(19):7266–7283
34. Zhang C, Cheng Y, Zhu L, Wu J (2016) Accuracy improvement of the immersed boundary-lattice Boltzmann coupling scheme by iterative force correction. Comput Fluids 124:246–260
35. Wu J, Shu C (2009) Implicit velocity correction-based immersed boundary-lattice Boltzmann method and its applications. J Comput Phys 228:1963–1979
36. Niu XD, Shu C, Chew YT, Peng Y (2006) A momentum exchange-based immersed boundary-lattice Boltzmann method for simulating incompressible viscous flows. Phys Lett A 354:173–182
37. Wu J, Shu C, Zhang YH (2010) Simulation of incompressible viscous flows around moving objects by a variant of immersed boundary-lattice Boltzmann method. Int J Numer Methods Heat Fluid Flow 62:327–354
38. Cheng Y, Zhu L, Zhang C (2014) Numerical study of stability and accuracy of the immersed boundary method coupled to the lattice Boltzmann BGK model. Commun Comput Phys 16:136–168
39. Cheng Y, Zhang H (2010) Immersed boundary method and lattice Boltzmann method coupled FSI simulation of mitral leaflet flow. Comput Fluids 39:871–881
40. Shu C, Liu N, Chew Y-T (2007) A novel immersed boundary velocity correction-lattice Boltzmann method and its application to simulate flow past a circular cylinder. J Comput Phys 226:1607–1622
41. Liu N, Peng Y, Liang Y, Lu X (2012) Flow over a traveling wavy foil with a passively flapping flat plate. Phys Rev E 85:056316
42. Lee P, Griffith BE, Peskin CS (2010) The immersed boundary method for advection–electrodiffusion with implicit timestepping and local mesh refinement. J Comput Phys 229:5208–5227
43. Fai TG, Griffith BE, Mori Y, Peskin CS (2014) Immersed boundary method for variable viscosity and variable density problems using fast constant-coefficient linear solvers II: theory. SIAM J Sci Comput 36:B589–B621
44. Huang H, Sukop M, Lu X (2015) Multiphase lattice Boltzmann methods: theory and application. Wiley, Hoboken
45. Guo Z, Shu C (2013) Lattice Boltzmann method and its applications in engineering. World Scientific, Singapore
46. Qian YH (1990) Lattice gas and lattice kinetic theory applied to the Navier-Stokes equations, PhD thesis. University Pierre et Marie Curie, Paris (1990)
47. Hou S, Zou Q, Chen S, Doolen G, Cogley A (1995) Simulation of cavity flow by the lattice Boltzmann method. J Comput Phys 118:329

48. He X, Chen S, Zhang R (1999) A lattice Boltzmann scheme for incompressible multiphase flow and its application in simulation of Rayleigh-Taylor instability. J Comput Phys 152:642–663
49. Wolf-Gladrow DA (2000) Lattice-gas cellular automata and lattice Boltzmann models—an introduction. Springer, Berlin
50. Succi S (2001) The lattice Boltzmann equation. Oxford Univ Press, Oxford
51. Luo LS (1998) Unified theory of the lattice Boltzmann models for nonideal gases. Phys Rev Lett 81:1618
52. Kraus J (2014) Optimizing a LBM code for compute clusters with Kepler GPUs. http://on-demand.gputechconf.com/gtc/2014/presentations/S4186-optimizing-lbm-code-compute-clusters-kepler-gpus.pdf
53. Valero-Lara P, Igual FD, Prieto-Matías Pinelli A, Favier J (2015) Accelerating fluid–solid simulations (lattice-Boltzmann & immersed-boundary) on heterogeneous architectures. J Comput Sci 10:249–261
54. Mawson M, Valero-Lara P, Favier J, Pinelli A, Revell A (2013) Fast fluid–structure interaction using lattice Boltzmann and immersed boundary methods. In: NVIDIA GPU Conference
55. Wu J, Cheng Y, Zhou W, Zhang C, Diao W (2016) GPU acceleration of FSI simulations by the immersed boundary-lattice Boltzmann coupling scheme. Comput Math Appl. doi:10.1016/j.camwa.2016.10.005
56. Bhaniramka P, Demange Y (2002) OpenGL volumizer: a toolkit for high quality volume rendering of large data sets. In: 2002 Symposium on Volume Visualization and Graphics, pp 45–53
57. Ahrens J, Geveci B, Law C (2005) ParaView: an end user tool for large data visualization. Visualization Handbook, Elsevier. ISBN 13:978-0123875822
58. Childs H, Brugger E, Whitlock B, Meredith J, Ahern S, Pugmire D, Biagas K, Miller M, Harrison C, Weber GH, Krishnan H, Fogal T, Sanderson A, Garth C, Bethel E, Camp D, Rübel O, Durant M, Favre JM, Navrátil P (2012) VisIt: an end-user tool for visualizing and analyzing very large data. In: High performance visualization—enabling extreme-scale scientific insight, pp 357–372
59. Bhatnagar PL, Gross EP, Krook M (1954) A model for collision processes in gases, I; small amplitude process in charged and neutral one-component system. Phys Rev 94:511
60. Bailey M, Cunningham S (2012) Graphics shaders theory and practice, 2nd edn. CRC Press, Boca Raton
61. Weiskopf D (2006) GPU based interactive visualization techniques. Springer, Berlin
62. Telea AC (2015) Data visualization principles and practice, 2nd edn. CRC Press, Boca Raton
63. Yu H, Wang C, Ma KL (2007) Parallel hierarchical visualization of large time-varying 3D vector fields. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, ACM, Nov 16, p 24
64. Xu C, Prince J (1998) Snakes, shapes, and gradient vector flow. IEEE Trans Image Process 7:359–369
65. Spencer B, Laramee RS, Chen G, Zhang E (2009) Evenly space streamlines for surfaces: an image based approach. Comput Graph Forum 28:1618–1631
66. Max N, Becker B, Crawfis R (1993) Flow volumes for interactive vector field visualization. In: Proceedings Visualization '93, pp 19–24