



GPU accelerated simulations of bluff body flows using vortex particle methods

Diego Rossinelli^a, Michael Bergdorf^a, Georges-Henri Cottet^b, Petros Koumoutsakos^{a,*}

^a Chair of Computational Science, ETH Zurich, CH-8092, Switzerland

^b Laboratoire Jean Kuntzmann, University of Grenoble and CNRS, BP 53, 38041 Grenoble, France

ARTICLE INFO

Article history:

Received 1 August 2009

Received in revised form 23 November 2009

Accepted 8 January 2010

Available online 20 January 2010

Keywords:

GPU

Vortex methods

Penalization

Bluff body flows

ABSTRACT

We present a GPU accelerated solver for simulations of bluff body flows in 2D using a remeshed vortex particle method and the vorticity formulation of the Brinkman penalization technique to enforce boundary conditions. The efficiency of the method relies on fast and accurate particle-grid interpolations on GPUs for the remeshing of the particles and the computation of the field operators. The GPU implementation uses OpenGL so as to perform efficient particle-grid operations and a CUFFT-based solver for the Poisson equation with unbounded boundary conditions. The accuracy and performance of the GPU simulations and their relative advantages/drawbacks over CPU based computations are reported in simulations of flows past an impulsively started circular cylinder from Reynolds numbers between 40 and 9500. The results indicate up to two orders of magnitude speed up of the GPU implementation over the respective CPU implementations. The accuracy of the GPU computations depends on the Re number of the flow. For Re up to 1000 there is little difference between GPU and CPU calculations but this agreement deteriorates (albeit remaining to within 5% in drag calculations) for higher Re numbers as the single precision of the GPU adversely affects the accuracy of the simulations.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Fast and accurate simulations of bluff body flows is of great interest to applications ranging from engineering aerodynamics to real time virtual reality simulations. Graphics processing Units (GPUs) present an energy efficient architecture for High Performance Computing (HPC) in flow simulations [11,39]. Besides fluid mechanics several researchers [31,41,18,13] have demonstrated that GPU accelerated simulations offer one to two orders of magnitude speedup over respective CPU implementations and they can assist in rapid-prototyping simulations to be launched on larger HPC systems.

The advent of simulations on GPUs was initiated in the computer graphics community where in turn fast simulations of fluid flows have been one of the first testing grounds for GPU accelerated physically based animations. A number of GPU based flow solvers have been proposed [14,30] based on Stam's "stable fluids" [40] and Lattice Boltzmann models [44,28]. Particle-based solvers have become popular in Computer graphics, as they bypass the CFL conditions often imposed by grid-based solvers and, in addition, do not require a grid to describe the flow around complex and deforming geometries. This robustness however, comes at a price: efficient Lagrangian algorithms are less straightforward to port to the GPU as they do not expose as much parallelism as grid-based methods. The most important issue in flow simulations using particles is the Lagrangian distortion of the computational elements that leads to inaccurate simulations. A number of ad-hoc

* Corresponding author. Tel.: +411 632 5258; fax: +411 632 1703.

E-mail address: petros@ethz.ch (P. Koumoutsakos).

techniques have been proposed to circumvent this difficulty, but in order to get a method that converges to the solution of the equations it discretizes, the maintenance of the smooth particle overlap is necessary [10,38,17].

Particle-mesh methods [22,9] for flow simulations employ a grid to remesh the particle locations when they get distorted. Furthermore, these methods can use the underlying grid in order to discretize the Laplacian operator and to solve the field equations [23], thus providing significant speedups over particle methods that perform these operations on the particles. Particle-mesh methods combine the accuracy and parallelism of grid-based methods with the robustness of particle methods to perform efficient simulations. These advantages come at the price of removing the grid-free label of particle methods. In order to circumvent these difficulties a number of particle methods have been proposed, including mesh aligned boundaries [24,25], mappings [9] and domain decomposition techniques [34]. These techniques are however geometry dependent and may not be readily extended to flows past complex and deforming geometries. In order to accommodate such geometries one may consider immersed boundary methods [35] which were originally developed in the context of vortex methods. In immersed boundary methods (see [32] and references therein) or in immersed interface [27,29], the solid is represented by a local modification of the governing equations for the fluid. A related simple and efficient approach to represent solid boundaries is the Brinkman penalization technique [2]. The penalization technique has been coupled with spectral methods and coherent vortex simulation methods [19], and adaptive wavelet collocation methods [43]. The method has been recently extended to provide a seamless formulation of flow-structure interaction problems combining particle methods and level sets [8]. The use of penalization techniques in particle methods simplifies the requirements for the structure of the grids used for remeshing and the discretization of the field operators. In turn this simplification enables the use of GPUs to carry out the computationally intensive parts of the simulations.

In this paper we exploit the simplicity and inherently parallel nature of Brinkman penalization to extend our GPU hybrid particle-mesh two-dimensional solver for unbounded domains [39] so as to handle flows with solid obstacles. The components of our solver are written in OpenGL and CUDA, which co-operate through the OpenGL/CUDA interoperability. Except for the OpenGL component that performs the particle-mesh operations, all the components can be written in any GPU framework: it can be CUDA, OpenCL, Brook+ (or OpenGL). We discuss the choice of the penalization parameters, the order of the penalization steps in a fractional step algorithm and the accuracy in time for integrating the penalization term. The method is validated on simulations of flows past an impulsively started cylinder and it is shown to provide orders of magnitude speedup over the same CPU based solver.

The paper is organized as follows. In Section 2 we describe the method and its GPU implementation in detail, and in Section 3 we validate our approach with simulations of the flow past an impulsively started cylinder for different Reynolds numbers. We conclude in Section 4 with performance results and a discussion of the limitations and strengths of this on-GPU flow solver.

2. Method

2.1. Vortex methods

We consider the vortical, incompressible, viscous flows as described by the Navier–Stokes equations in the $\mathbf{u} - \omega$ formulation:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} + \mathbf{U}_\infty) \cdot \nabla \omega = \nu \Delta \omega, \quad (1)$$

$$\Delta \Psi = -\nabla \times \mathbf{u} = -\omega, \quad (2)$$

$$\lim_{\mathbf{x} \rightarrow \infty} \mathbf{u} = \mathbf{0}. \quad (3)$$

As the formulation is two-dimensional, the term $\omega \cdot \nabla \mathbf{u}$ is omitted.

We discretize these equations using remeshed vortex methods [9,23]. The vorticity field is approximated using a linear superposition of particles,

$$\omega_\epsilon^h(\mathbf{x}, t) = \sum_p \Gamma_p(t) \zeta_\epsilon(\mathbf{x} - \mathbf{x}_p^h(t)), \quad (4)$$

where Γ_p denotes the particle strengths, $\mathbf{x}_p(t)$ the particle locations and ζ_ϵ is the reconstruction kernel. The discretization Eq. (1) yields a set of ordinary differential equations for the particle positions and their circulation strength:

$$\frac{d\Gamma_p}{dt} = \nu \Delta \omega, \quad (5)$$

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{u}(\mathbf{x}_p, t), \quad (6)$$

$$\text{with } \mathbf{u} = -\nabla \times \Delta^{-1} \omega. \quad (7)$$

The particle trajectories and strengths are obtained by integrating these equations, thus enabling the automatic adaptivity of the computational elements as dictated by the flow map. Vortex particle methods have the advantage of no linear CFL

restrictions as well as negligible numerical dissipation and dispersion errors. For nonlinear stability the time step must satisfy $\delta t \leq C_1 \|\nabla \otimes \mathbf{u}\|_\infty^{-1} \approx C_2 \|\omega\|_\infty^{-1}$, where C_1 and C_2 are constants depending on the time-integration scheme.

However the adaptation of particle methods comes at the expense of the regularity of the particle distribution as particles adapt to the gradient of the flow field. The numerical analysis of vortex methods shows that the truncation error of the method is amplified exponentially in time, at a rate given by the first-order derivatives of the flow that are related to the amount of flow strain. In practice, particle distortion can result in the creation and evolution of spurious vortical structures due to the inaccurate resolution of areas of high shear and the resulting inaccurate approximations of the differential operators (e.g. for diffusion) [23]. To remedy this situation, location-processing techniques reinitialize the distorted particle field onto a regularized set of particles and simultaneously accurately transfer the particle quantities. The accuracy of remeshing has been thoroughly investigated in [22] and it was shown to introduce numerical dissipation that is well below the dissipation introduced by the temporal and spatial discretizations. An efficient way to regularize the particles is by reinitialising the particle positions on a cartesian grid and recomputing the transported quantities with a particle-mesh operation [23,6]. The presence of the grid offers an additional advantage in terms of computational speed for the hybrid grid-particle methods as it can be used to solve for the diffusion equation [7] using a finite difference scheme when particles have been remeshed on the grid. This differentiation accelerates significantly the computation of differential operators for particle methods over techniques such as SPH as it only requires a finite difference stencil with 5/7 points versus about 25/125 points (for a Gaussian kernel) in 2D/3D simulations, respectively. A further advantage of this approach is the capability to introduce multiresolution in particle methods [3,4].

2.2. Penalization

The penalization technique [1] is used to couple the flow field with the obstacles in the domain. The solid obstacle covers a region Ω_s and has a velocity \mathbf{U}_s and it is modeled as a porous medium [5] with vanishing porosity. The computational porosity of the obstacle is controlled by two quantities: the characteristic function χ_S , which localizes the obstacle by taking the value of 1 inside the obstacle and zero outside, and the factor $\lambda \gg 1$ that is inversely proportional to the porosity of the obstacle. The Brinkman-penalized Navier–Stokes equations in $(\mathbf{u} - \omega)$ [8] are expressed as:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} + \mathbf{U}_\infty) \cdot \nabla \omega = \nu \Delta \omega + \lambda \nabla \times [\mathbf{U}_s - (\mathbf{u} + \mathbf{U}_\infty)] \chi_S. \quad (8)$$

For $\lambda \rightarrow \infty$ the velocity of the fluid at the obstacle boundary will be \mathbf{U}_s . However increasing λ leads to a stiffer set of equations and care should be taken for choosing the correct range of this value. We obtained good results in the range of $\lambda \in [10^4, 10^5]$.

2.3. Particle-based flow solver

The evolution of the flow is considered in a fractional step algorithm where at each time step the vorticity field is penalized near the boundaries, diffused (according to $\nu \Delta \omega$) and convected (according to $(\mathbf{u} + \mathbf{U}_\infty) \cdot \nabla \omega$). The three-steps splitting algorithm is expressed as:

1. The penalization term in the vorticity frame: $\frac{\partial \omega}{\partial t} = \nabla \times [\mathbf{U}_s - (\mathbf{u} + \mathbf{U}_\infty)]$,
2. Diffusion of the vorticity: $\frac{\partial \omega}{\partial t} = \nu \Delta \omega$,
3. Lagrangian advection of the vorticity: $\frac{d\mathbf{x}}{dt} = \mathbf{u}$.

We can equivalently express the splitting algorithm so as to obtain second order accuracy in the time discretization. With particles $S^n = \{(\mathbf{x}_{p,\text{grid}}, \omega_p^n)\}$ located in every point of a regular equidistant grid at time t^n we list below the steps of the solver so as to obtain a new set of particles $S^{n+1} = \{(\mathbf{x}_{p,\text{grid}}, \omega_p^{n+1})\}$ at the grid points at time t^{n+1} .

- (1) Compute the velocity on the grid with the condition that the velocity should go to zero at infinity (unbounded domain). \mathbf{u}_{pen} denotes the velocity after penalization, and ω_{pen} its corresponding vorticity.

$$\Delta \Psi = -\omega^n, \quad (9)$$

$$\mathbf{u}^n = \nabla \times \Psi. \quad (10)$$

- (2a) Integrate the penalization implicitly (first order), and compute the corresponding vorticity correction on the grid.

$$\frac{\mathbf{u}_{\text{pen}}^* - \mathbf{u}^n}{\delta t} = \lambda [\mathbf{U}_s - (\mathbf{u}_{\text{pen}}^* + \mathbf{U}_\infty)] \chi_S^h, \quad (11)$$

$$\omega_{\text{pen}}^* = \nabla \times \mathbf{u}_{\text{pen}}^*. \quad (12)$$

- (2b) If necessary, measure the drag, and add the free stream velocity \mathbf{U}_∞ to the solution $\mathbf{u}_{\text{pen}}^*$.

$$f_{\text{drag}} = \lambda \int \chi_S \mathbf{u}_{\text{pen}}^* d\mathbf{x} \quad (13)$$

$$\mathbf{u}_{\text{pen}}^n = \mathbf{u}_{\text{pen}}^* + \mathbf{U}_\infty. \quad (14)$$

(3) Add the contribution of the vorticity diffusion with an explicit second order update.

$$\omega^{*,n} = \omega_{\text{pen}}^n + \frac{\delta t}{2} \Delta \omega_{\text{pen}}^n, \quad (15)$$

$$\omega_{\text{particles}}^n = \omega_{\text{pen}}^n + \delta t \Delta \omega^{*,n}. \quad (16)$$

(4) Advection the particles with velocity $\mathbf{u}_{\text{pen}}^n$ with a second order accurate explicit time stepper. To evaluate $\mathbf{u}_{\text{pen}}^n$ at the particle locations we interpolate it from the grid point locations with the reconstruction kernel ζ_ϵ .

$$\mathbf{x}_p^* = \mathbf{x}_{p,\text{grid}} + \frac{\delta t}{2} \mathbf{u}_{\text{pen}}^n(\mathbf{x}_{p,\text{grid}}), \quad (17)$$

$$\mathbf{x}_p^{n+1} = \mathbf{x}_{p,\text{grid}} + \delta t \mathbf{u}_{\text{pen}}^n(\mathbf{x}_p^*). \quad (18)$$

(5) Interpolate the vorticity carried by the particles onto grid point locations using the remeshing kernel M .

$$\omega_q^{n+1} = \sum_p \omega_p M\left(\frac{1}{h} [\mathbf{x}_p^{n+1} - \mathbf{x}_{q,\text{grid}}]\right). \quad (19)$$

3. Computational details

3.1. A periodic Poisson solver for an unbounded domain

In order to solve the Poisson equation for an unbounded domain, we adopt the extended domain technique described in Hockney and Eastwood [16] so as to benefit from the efficient implementation of FFTs on GPUs. In this technique, a vorticity field initialised on a domain of $n \times n$ elements is copied into the first quadrant of a $2n \times 2n$ grid and the remaining elements of the grid are zero padded. We then perform a Fourier transform of this “extended” right-hand side. The transformed field is multiplied with a $2n$ Green’s function which was previously been sampled and mirrored in the physical space. Finally, the velocity is obtained by taking the curl of the Fourier inverse-transform of the computed data in the first quadrant. Besides its spectral accuracy, this approach is well suited for GPGPU computing due to its compactness, data-parallelism and simplicity. This speed-up comes at the expense of increased memory requirement as solving a Poisson equation for a grid size of $n \times n$ it is 4 times slower than the respective periodic solver and 8 times more memory consuming.

Furthermore, the accuracy of this method is only assured if the vorticity at the boundary of the grid is zero. In order to ensure zero vorticity at the grid boundary, we have to fade out the vorticity field at the end of the computational domain. We therefore have the vorticity smoothly “decay” in space using the following function:

$$W_{\text{decay}}(x, x_{\text{end}}, L) = \frac{1}{2} \cos\left(\pi \frac{x - (x_{\text{end}} - L)}{L}\right) + \frac{1}{2}, \quad (20)$$

where x is the location where we would like to evaluate the decay, x_{end} is the physical position of the downstream boundary and L is the physical length in which the fading out is performed. In this work the downstream grid boundary is always located at $x_{\text{end}} = 1$ and we used $L = 10\delta x$.

3.2. Reconstruction and remeshing kernels

The grid-particle and particle-grid interpolations, including the remeshing step at the end of each time step (see Eq. (19)), are performed by using the $M'_4(x, y) = M'_4(x) \cdot M'_4(y)$ kernel [23] which is given as:

$$M'_4(x) = \begin{cases} 0 & \text{if } |x| > 2, \\ \frac{1}{2}(2 - |x|)^2(1 - |x|) & \text{if } 1 \leq |x| \leq 2, \\ 1 - \frac{5}{2}x^2 + \frac{3}{2}|x|^3 & \text{if } 1 \geq |x|. \end{cases} \quad (21)$$

3.3. Smoothing of the penalisation function

The penalisation function χ_s is mollified χ_s^ϵ based on a signed distance transform φ of the obstacle:

$$\chi^\epsilon(\mathbf{x}) = \frac{1}{2} + \frac{1}{2} \cos \pi \alpha(\varphi(\mathbf{x})), \quad (22)$$

$$\alpha(r) = \begin{cases} 0 & r < r_0 - \epsilon/2 \\ 1 & r > r_0 + \epsilon/2 \\ \frac{1}{2} + \frac{x-r_0}{\epsilon} & r \in [r_0 - \epsilon/2, r_0 + \epsilon/2] \end{cases}, \quad (23)$$

where ϵ is the mollification parameter and r_0 is the radius where the χ^ϵ assumes the value of $\frac{1}{2}$. For flow past cylinders we use $\chi_{\text{cylinder}}^\epsilon$ and a signed distance function such that χ_s^ϵ conserves the zeroth moment of the original characteristic function:

$$\varphi(\mathbf{x}) = |\mathbf{x} - \mathbf{x}_c| - r_0, \tag{24}$$

$$r_0 = \frac{(D^2 \pi^2 - \epsilon^2 (\pi^2 - 8))^{1/2}}{2\pi}, \tag{25}$$

where D is the diameter of the cylinder and \mathbf{x}_c denotes its center. We use $\epsilon = 2\sqrt{2}\delta x$, δx being the grid spacing.

4. GPU implementation

The present GPU solver uses textures that represent grids and particles, and employs kernels and shaders for processing the data and producing other textures. Fig. 1 depicts the global workflow of the solver.

4.1. Particle-mesh operations

A key aspect of the hybrid particle method is the interpolation between grids and particles (see Eq. (19)). In order to perform these interpolations efficiently on the GPU we use the technique explained in [39], which employs *point-sprite* primitives. Those primitives allow the use of points rather than quads and are able to generate texture coordinates which are interpolated across the point. Enabling the blending mode and computing the single contributions in the fragment shader, allows us to obtain the desired interpolated quantities on the grid very efficiently, as the GPU sums each contribution from every particle to the respective grid points in the framebuffer.

4.2. CUFFT-based poisson solver

We solve the Poisson equation with unbounded boundary conditions as described above using CUFFT (version 2.2/2.3) [33], the CUDA-based Fourier transform library. The GPU solver thus relies on the interoperability between CUDA and

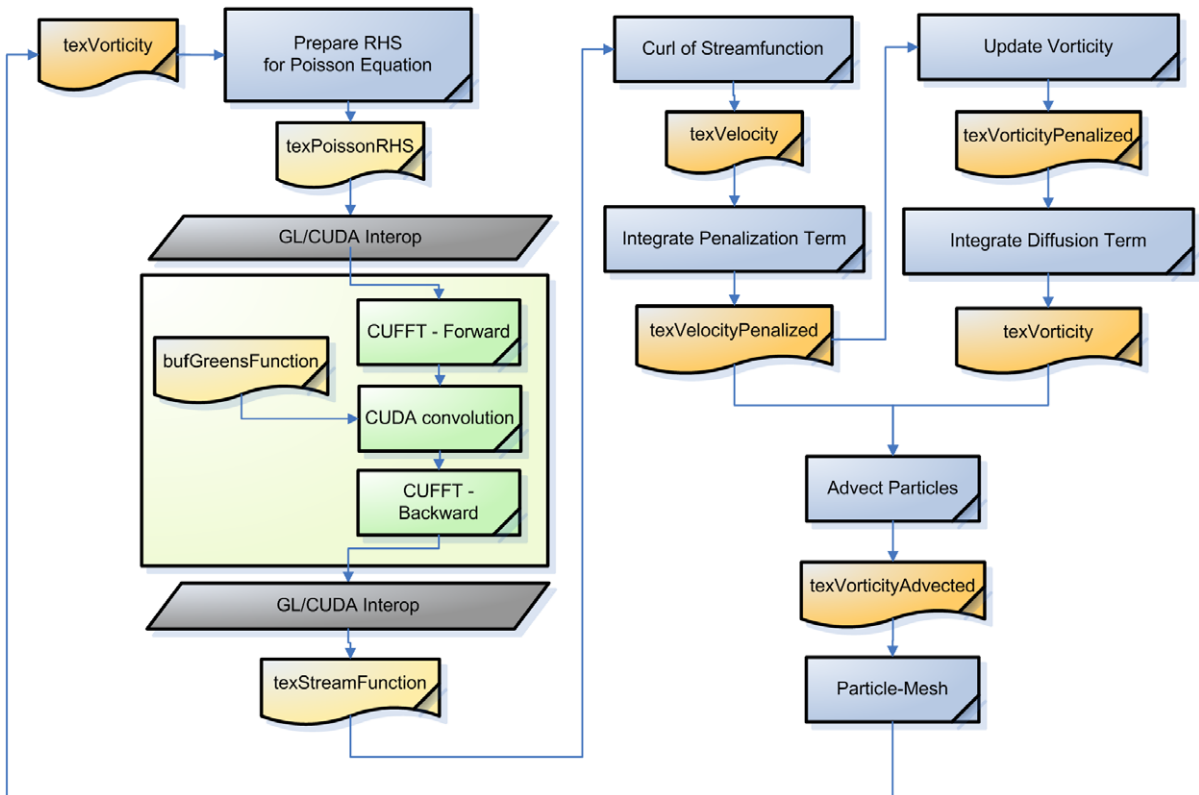


Fig. 1. Workflow of the GPU fluid solver. Simple textures (grids and particles) are depicted in orange. Textures that are close to the CUDA part are associated with GPU buffer objects. Shaders/kernels are depicted in blue/green. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

OpenGL, which is established by CUDA-(un)registering and CUDA-(un)mapping the OpenGL buffers (`pixel_buffer_object`, `vertex_buffer_object`). An OpenGL buffer, once it is CUDA-registered, can be CUDA-mapped and passed to a kernel through a buffer pointer. The vorticity is represented as a 2D texture of size $n \times n$. To find the velocity (and therefore solve the Poisson equation) we need to do the following:

1. Set a floating point texture of size $2n \times 2n$ as render-target (or framebuffer object). Draw the vorticity texture on the first quadrant, zero the remaining quadrants. Then perform a pixels read-back of the render-target into a pixel buffer object (pbo) of size `sizeof(float)*2*n*2*n`.
2. CUDA-register the pbo, CUDA-map it in order to obtain the buffer pointer.
3. Perform a forward CUFFT-2D $2n \times 2n$ of the vorticity pbo.
4. Compute the streamfunction in Fourier space by multiplying the fourier transformed vorticity with the Green's function \hat{G} in Fourier space.
5. Perform a backward CUFFT-2D $2n \times 2n$ of the streamfunction buffer.
6. Transfer the streamfunction pbo into the streamfunction texture. Then set the velocity texture as render-target and render the curl of the streamfunction texture (only the first quadrant).

4.3. Particle advection

Due to the non-periodic setting of our computational domain, particles can leave the domain. During particle-grid interpolations the contributions of particles that do not lie in the computational domain are simply discarded; this is handled straightforwardly by the geometry shader. To further improve the performance of the particle advection, in the present work we use a second order time stepper, which starts from particles lying on grid point locations. We use this fact to perform the advection step in one rendering pass (the previous solver was performing one pass per step in the Runge–Kutta integration), so that we need perform grid-particle interpolation operations only once.

4.4. Diffusion step in one pass

In terms of texture cache misses and synchronization, the overhead cost associated to the launching of a grid (CUDA) or preparing a rendering pass (OpenGL) can be significant. We therefore aim at maximizing the work per rendering/computing pass. We achieve this by concatenating the subsequent algorithm steps into one pass. In this work we recast the two-stage Runge–Kutta advection and diffusion steps into two single-pass processes. While recasting advection is natural for the advection pass (see above), the diffusion step needs some symbolic manipulations. The single-pass process for performing a second order time-accurate diffusion can be expressed as:

$$\omega_{ij}^{n+1} = \frac{\alpha^2}{2} \left[\omega_{i-2j}^n + \omega_{i+2j}^n + \omega_{ij-2}^n + \omega_{ij+2}^n \right] + \alpha^2 \left[\omega_{i+1j+1}^n + \omega_{i-1j+1}^n + \omega_{i+1j-1}^n + \omega_{i-1j-1}^n \right] \\ + \alpha(1 - 4\alpha) \left[\omega_{i+1j}^n + \omega_{i-1j}^n + \omega_{ij+1}^n + \omega_{ij-1}^n \right] + (1 - 4\alpha + 10\alpha^2)\omega_{ij}^n,$$

where $\alpha = \nu \delta t / \delta x^2$. Considering that the values of ω are fetched from a texture, the benefits of the single-pass approach are double. The first benefit is that in the best case there is a reduction by a factor of 2 of the overall cache misses in the texture cache. The second benefit is that by doing one pass instead of two, one avoids the synchronization barrier that would have been present in the case of two passes.

5. Results

We use the drag and lift coefficients of the flow as diagnostics for our simulations. We note that the drag coefficient for impulsively started flows, is a challenging quantity, as it exhibits a near singular character at the onset of the motions and it is very sensitive to the detailed vortical structures in the vicinity of the body, in particular at higher Re numbers. We then analyze the discrepancies that are present between the GPU and CPU simulations, and show the best results obtainable with the employed GPUs.

5.1. Diagnostics

The Reynolds number (Re) of the flow is defined based on the diameter of the cylinder (D), as:

$$Re = \frac{U_\infty D}{\nu}, \quad (26)$$

and the time is nondimensionalized based on the radius ($R = D/2$) of the cylinder as:

$$T = \frac{U_\infty t}{R}. \quad (27)$$

The drag and lift coefficients (c_D and c_L , respectively) of the body are given by:

$$c_D = \frac{2F_{b,0}}{U_\infty^2 D}, \quad c_L = \frac{2F_{b,1}}{U_\infty^2 D}, \quad (28)$$

where \mathbf{F}_b is the force that the body exerts on the fluid. All the results we show here were obtained by simulations using $U_\infty = 0.1$, $\lambda = 10^4$ and a *LCFL* between 0.1 and 0.01. The flow past an impulsively started cylinder is used as a benchmark as it can severely challenge the accuracy of the method due to the fine vortical structures developed at the boundary and the singular behavior of the drag coefficient.

5.2. Discrepancies between CPU and GPU

In this subsection we analyze the difference in results between three different solvers: CPU solver in double precision, CPU solver in single precision and a GPU solver in single precision. In these studies we have used the same computational settings for all three solvers. We demonstrate the differences between the various solvers by comparing the drag and lift coefficients as well as the vorticity field of the flows in various time points.

5.2.1. Discrepancy in the drag and lift coefficients

Fig. 2 shows the drag and lift for $Re = 100$ and $Re = 300$ until $T = 250$, performing about 50,000 simulation steps with 1024×1024 particles. At $Re = 100$ we observe that after the start of the instability, the GPU results match well those of the CPU. The instability starts at the same time $T \sim 50$ for the GPU and the CPU solver in single precision, whereas with the double precision CPU solver, the instability is triggered at $T \sim 150$. In the case of $Re = 300$, the drag/lift coefficients are practically the same for all three solvers. However the values in the instability phase are different, especially for the CPU solver in double precision.

Fig. 3 shows the evolution of the drag coefficient at higher Reynolds numbers: $Re = 1000, 5500, 9500$. These plots were obtained by performing around 30,000 steps with 1024×1024 particles. The discrepancy between the GPU and the double precision CPU solver becomes progressively higher by increasing the Reynolds number. We observe that at $Re = 1000$ the three solvers give almost the same time-evolution of the drag. The average (over time) difference between the GPU and the double precision is 0.2%, whereas the discrepancy between the CPU solver in double and single precision is 0.1%. At

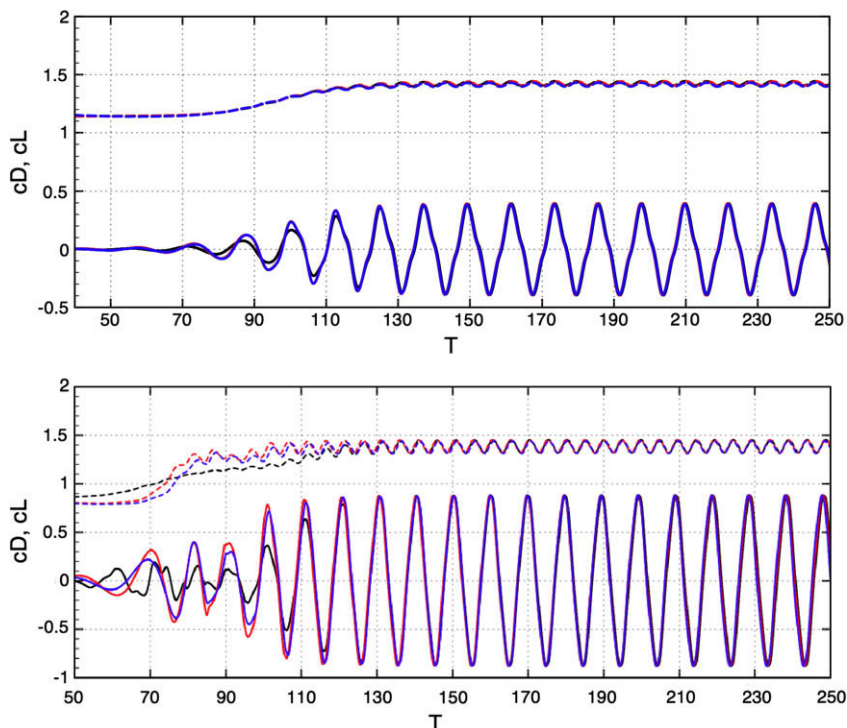


Fig. 2. Drag (dashed lines) and lift (contiguous lines) coefficients computed on the CPU in single and double precision (black and red) and on the GPU (blue), for the same computational settings, at different Reynolds number: 100 (top), 300 (bottom). As in the case $Re = 100$ and $Re = 300$ the instability starts at different times, the different plots were manually aligned (in time). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

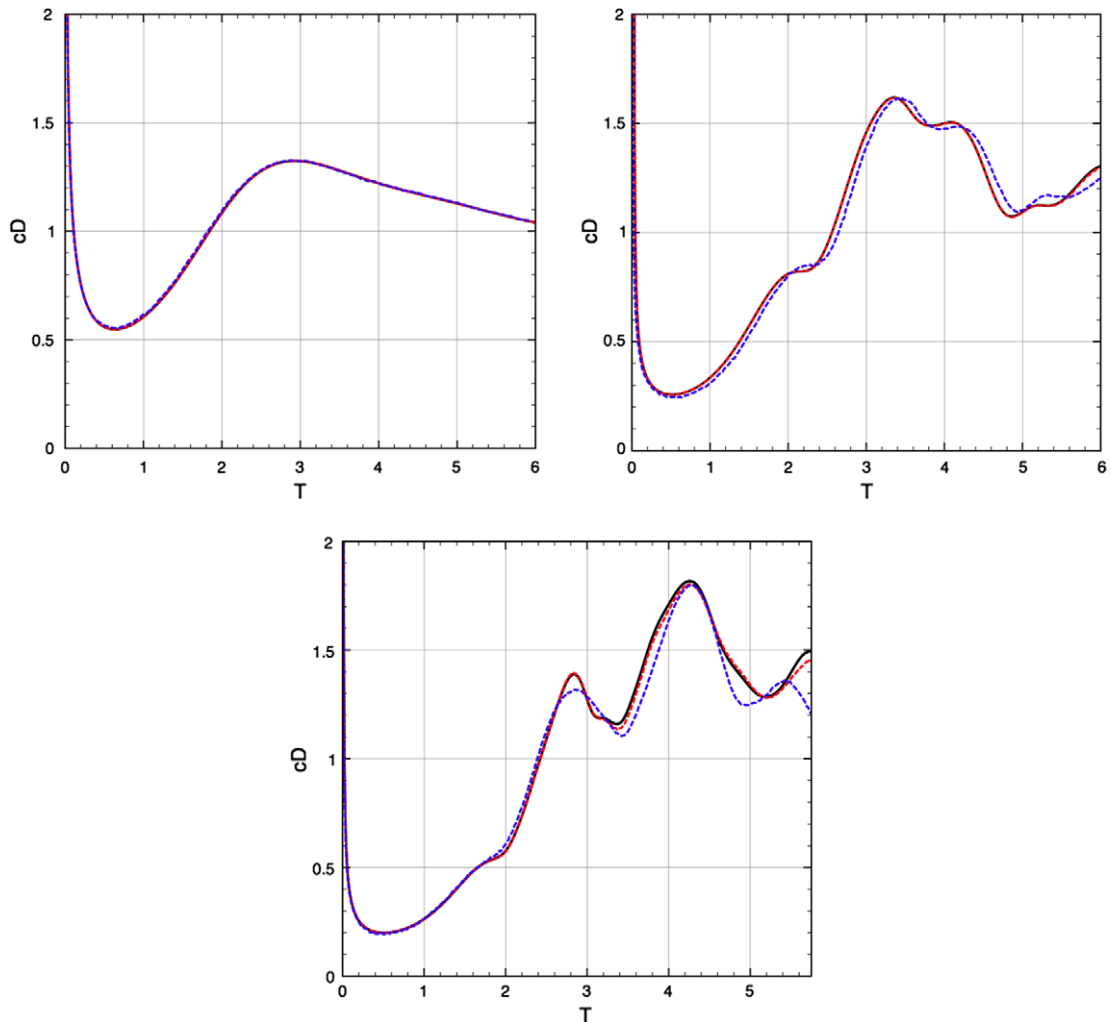


Fig. 3. Evolution of drag/lift coefficients at $Re = 1000$ (left), $Re = 5500$ (right), $Re = 9500$ (bottom) for the CPU in double precision (black), single precision (red) and GPU solver (blue) with the same computational settings. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

At $Re = 5500$ the GPU solver produces a drag coefficient that in average differs by 2.2% with respect to the double precision CPU solver, whereas the average discrepancy between the CPU solver in single precision and double precision is less 0.1%.

At $Re = 9500$ the CPU-computation in double precision differ substantially from the CPU single precision one, meaning that the accuracy of the finite arithmetic is a key player. It is therefore expected that the discrepancy of the GPU is even bigger, given the sources of inaccuracies illustrated in Fig. 7. The discrepancy of the GPU with respect to the double precision CPU solver is 2.8% and the float-double precision discrepancy on the CPU is 0.6%.

5.2.2. Discrepancy in vorticity

Fig. 4 illustrates contour lines of the vorticity at $Re = 100, 550$ for both GPU and double precision CPU solvers after 20,000 simulation steps ($T = 180$). The contour lines are overlapping almost everywhere.

Fig. 5 illustrates contour lines of the vorticity at $Re = 1000$, at the nondimensional time $T = 6$, after 20,000 iteration steps for the GPU and double precision CPU solver. The proximity of the blue/black lines show that the two vorticity fields are in good agreement. The contour lines of the vorticity at $Re = 9500$ are shown in Fig. 6 for the GPU and the double precision CPU solver. The contour lines are computed at time $T = 2.5, 3.5, 6.0$ after roughly 15,000, 25,000, 35,000, respectively simulation steps. At time $T = 2.5$ there is a good agreement between the GPU and the CPU lines. This agreement decreases at $T = 3.5$ where the downstream vorticity assumes two similar but different displacements. After 35,000 simulation steps, at time $T = 6.0$, we clearly observe some differences in the position of the two vortices in the farthest part of the wake and also in the vortices close to the body.

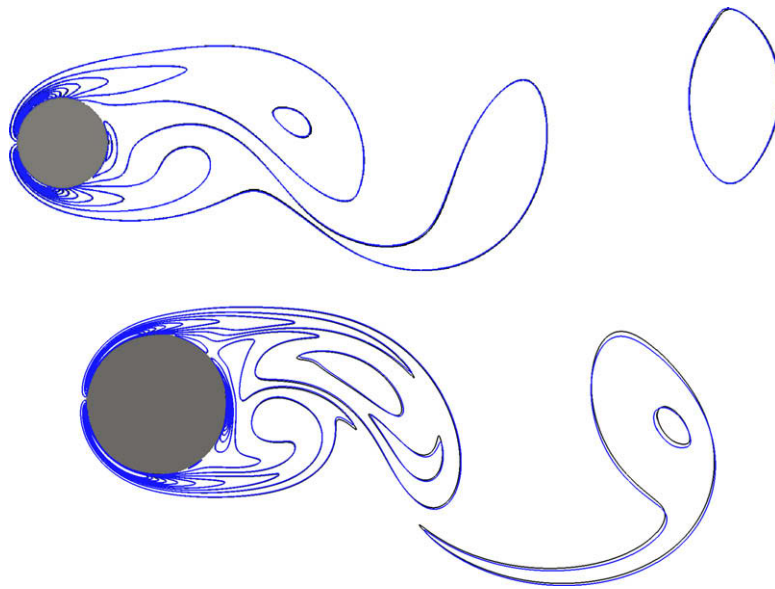


Fig. 4. Vorticity contour of the GPU results (blue) and CPU-double precision results (black), at Re 100 (top) and Re 550 (bottom) after roughly 20,000 simulation steps ($T = 180$). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

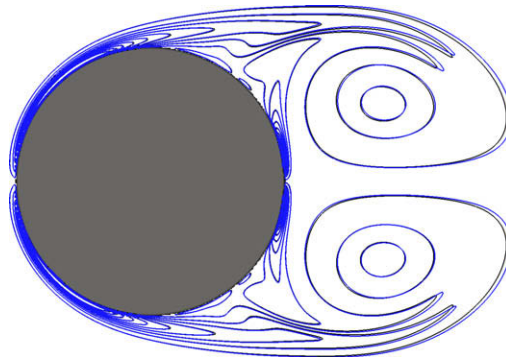


Fig. 5. Vorticity contour of the GPU results (blue) and CPU-double precision results (black), at Re 1000, $T = 6$ after 20,000 simulation steps. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

5.2.3. Discrepancy in the solver substeps

Fig. 7 illustrates the difference ($\|\cdot\|_\infty$ norm) of the substeps of the GPU solver with respect to the double precision CPU solver at $Re = 100, 9500$ for the first few thousands of simulation steps.

Firstly, we observe that the Reynolds number does not affect the ranking of the discrepancies. A second observation is that the magnitudes of the discrepancies do not strongly depend on the Reynolds number, and are of the order of 10^{-5} . The third observation is that the biggest source of discrepancy happens in solving the Poisson equation. The plot in Fig. 8 shows that both CPU and GPU exhibit the expected convergence (second order) of the Poisson solvers up to grids of 1024×1024 (i.e. 512×512 particles). For larger grids both GPU and CPU (single precision) are relatively inaccurate and in particular the GPU computations.

In order to investigate which part of the GPU Poisson solver was the most inaccurate, we replaced the on-GPU convolution and curl computation with a computation in double precision on the CPU. As this replacement did not help in improving the accuracy, we claim that the source of inaccuracy is inside CUFFT (version 2.2 and 2.3). This was also confirmed by running CUFFT in emulation mode: we obtained the same convergence plot as the one obtained with the CPU FFT-based Poisson solver in single precision. Further reading about the sources of inaccuracy of CUFFT can be found in the work of Govindaraju et al. [12].

5.3. Impulsively-started cylinder: Re 40–9500

The goal of this section is to show the *best* results that one can obtain with the CPU and the GPU solver, regardless of the computational settings. We compare the results (GPU and CPU) with results obtained with different methods in the

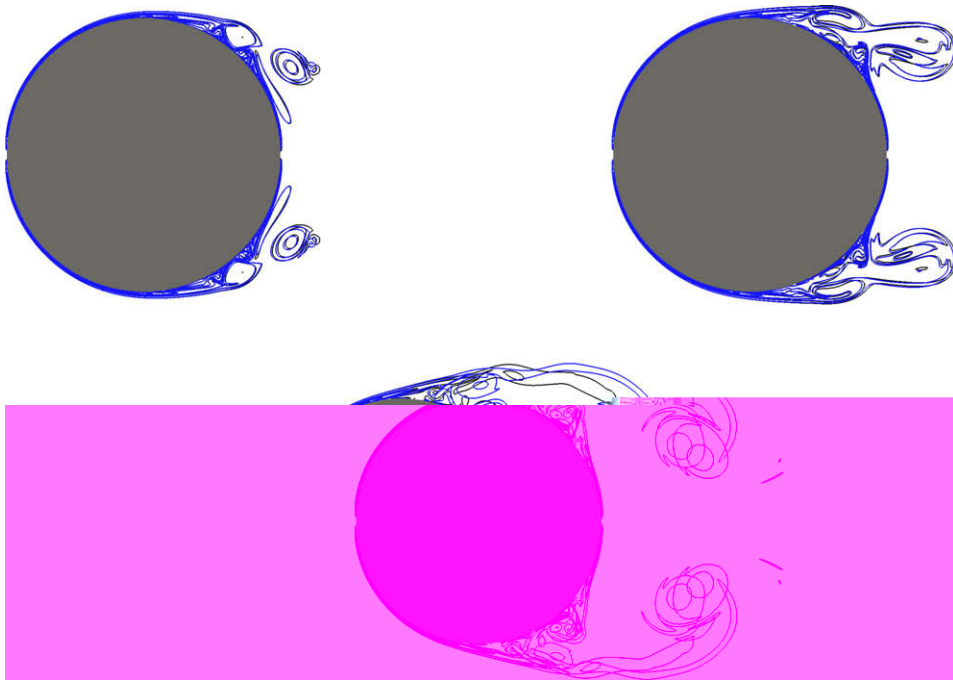


Fig. 6. Vorticity contour of the GPU results (blue) and CPU-double precision results (black), at Re 9500, $T = 2.5, 3.5, 6$. (top-left, top-right, bottom). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

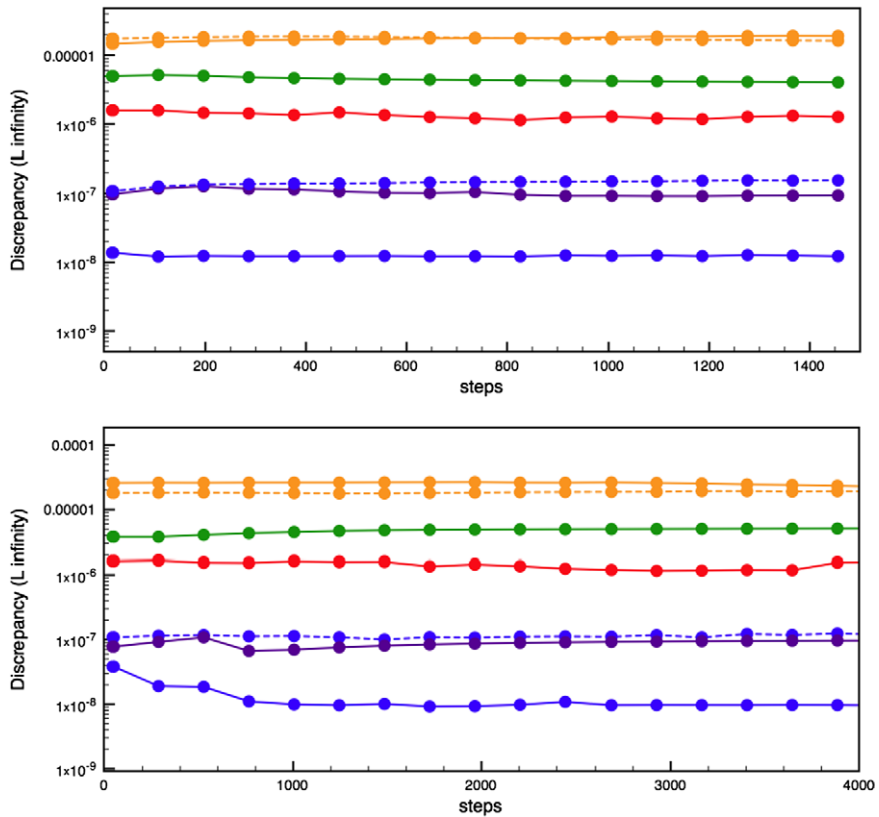


Fig. 7. Discrepancy between the computation in double precision of the CPU and the GPU for the substeps of algorithm, for different Reynolds numbers: 100 (top) and 9500 (bottom). The sources of discrepancy are in: the Poisson solver (orange, dashed line for the u-component), the particles advection step (green), the particle-mesh step (red), the diffusion step (violet) and the penalization step (blue, dashed-line for the u-component). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

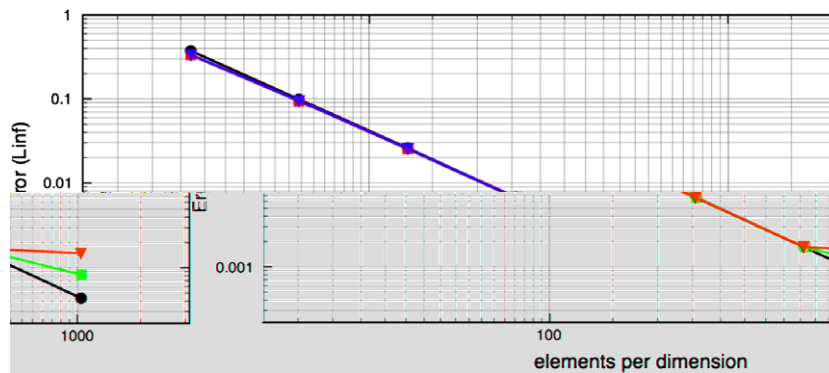


Fig. 8. Convergence study of the Poisson solvers: FFTW-double precision based (black) FFTW-single precision based (red), CUFFT-based (blue) solvers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

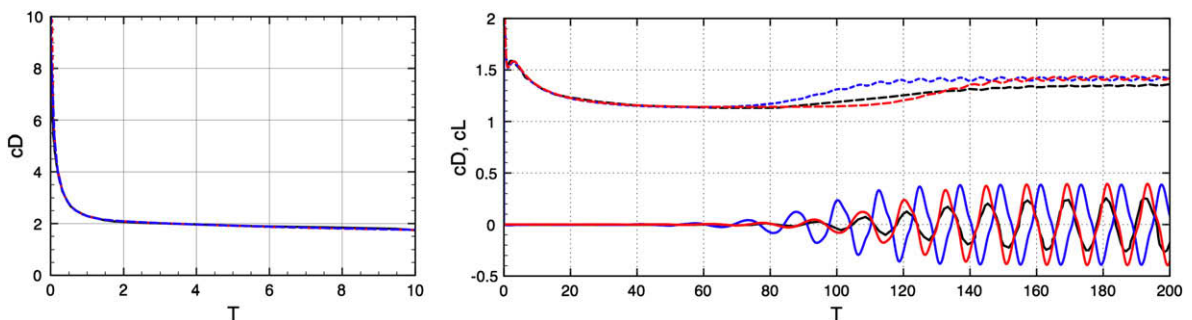


Fig. 9. Drag Coefficient for an impulsively started circular cylinder. $Re = 40, 100$ (left, right) for the CPU solver (red), GPU solver (blue), and the reference drag [24] (symbols) ($Re = 40$), [20] ($Re = 100$). Note that these results are obtained with computational settings of the GPU that are different with respect to the ones of the CPU.

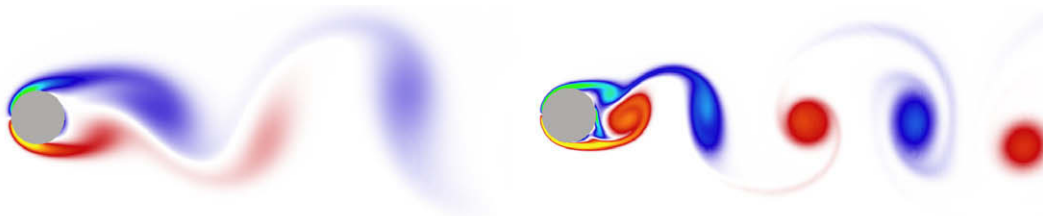


Fig. 10. Vorticity at $Re = 100, 500$ (left, right) after 20,000 simulation steps. Blue/red colors denote high negative/positive vorticity. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

literature. The drag of the CPU was obtained by using 4096×1024 particles (using more particle did not change significantly the results), whereas the GPU solver was using 1024×1024 particles because a CUFFT limitation. In theory CUFFT is capable to process data up to 16384 elements per dimensions [33], however in practice this limit depends on the GPU model. The GPUs that we employed, GeForce 9800 GX 2, GeForce 8000 GTS, and QuadroFX 5600, only support transform up to 2048×2048 .

$Re = 40$. At this Re the flow is known to reach a steady state. Our goal was to terminate the computation when the drag coefficient (Fig. 9) has closely approached the value it would assume after infinite time has elapsed. Our computation of drag coefficient matches the experimental value of 1.68 [42].

$Re = 100$. Fig. 10 shows the vorticity at $T = 150$ when the von Kármán vortex-street is well established. Fig. 9 show the lift and the drag coefficients, compared to the reference solution by Kevlahan and Vasyliev [20]. The average drag coefficients was 1.35 for the CPU solver and 1.41 for the GPU solver. The amplitude of the lift were 0.38 and 0.42 for the CPU- and the GPU-solver, respectively.

If we compare the drag obtained with the double precision CPU solver with 1024×1024 particles (Fig. 2) with the one with 4096×1024 particles (10) we note a substantial difference. This is explained by the fact that the vorticity wake of the

Table 1

Average of the drag coefficient, amplitude of the lift coefficient and Strouhal number are reported for CPU/GPU simulations at different Reynolds numbers.

Re	c_D	c_L	S_r	Source
40	1.59–1.69	0	0	[24]
40	1.68	0	0	present-CPU
40	1.68	0	0	present-GPU
100	1.29–1.37	0.29–0.31	0.166–0.176	[20]
100	1.35	0.33	0.166	present-CPU
100	1.41	0.34	0.170	present-GPU
500	1.44–1.45	1.18–1.20	0.226–0.229	[36]
500	1.45	1.20	0.230	present-CPU
500	1.46	1.3	0.220	present-GPU

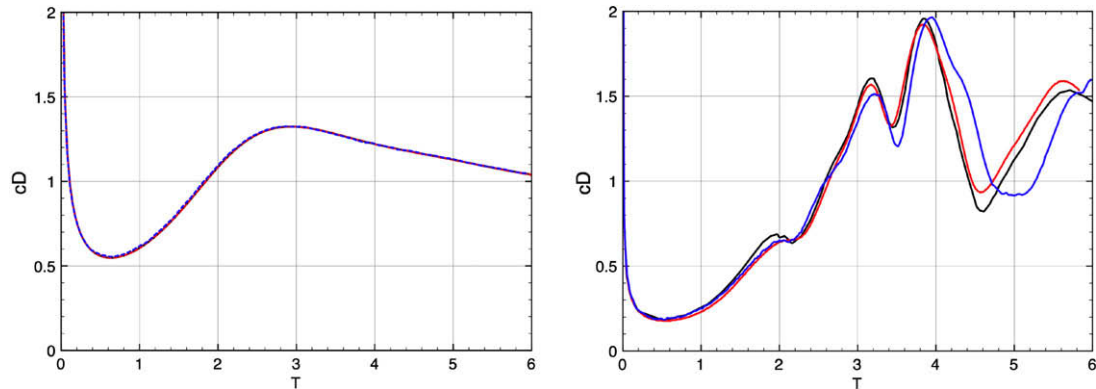


Fig. 11. Comparison of the drag coefficient of an impulsively started cylinder at $Re = 1000, 9500$ (left, right respectively) as computed by several numerical methods. Red and blue lines were obtained with the CPU- and GPU-solvers whereas the reference computation is denoted with a black line [24]. Note that these results are obtained with computational settings of the GPU that are different with respect to the ones of the CPU. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

cylinder still assumes strong values far from the cylinder surface ($x \sim 10D$). In order to compute the drag accurately, a grid that is big enough to include the strongest part of the wake is necessary. Therefore using a computational settings of 4096 particles instead of 1024 (hence considering a longer wake) in the x -direction can substantially improve the results.

The reference drag and the drag of the CPU solver at $Re = 100$ in Fig. 10 have around 1% discrepancy, which is not big if one considers that the reference drags in the literature differ by 6% [37,15,26]. Regarding the lift coefficient, the CPU solver has a discrepancy of 6% in the amplitude with respect to the reference solver. The reason is that the lift coefficient is more sensitive to the wake than the drag coefficient, because of the horizontal displacement of the vortices in the wake. The CPU solver cannot consider the entire wake of the object, whereas the reference solver used an adaptive solver, which is able (at least in theory) to track all the vortices in the wake. We note that in the literature lift coefficients differ by 6% [37,26].

$Re = 500$. As the Re increases the flow-structure becomes more complicated and the vortex cores in the wake are more localized with respect to $Re = 100$. Fig. 10 (right) depicts the vortex shedding at $T = 80$ and $Re = 500$. The reference drag and lift coefficients at $Re = 500$ are the same as the ones obtained with the CPU solver, illustrated in Table 1. While the drag coefficient obtained with the GPU solver is very similar to the one obtained with the CPU solver (0.6% discrepancy), the CPU-GPU discrepancy in the lift coefficient is much larger (8%). We attribute this discrepancy to the truncation of the wake vortices along the centerline of the domain (see Fig. 10 (right)). Vortices displaced along the centerline of the domain while they have little effect on the drag they contribute more significantly to the lift coefficient. Hence the truncation of the wake has a qualitatively greater impact on the lift than on the drag coefficient.

$Re = 1000$. The parameter to observe here is the drag coefficient shown in Fig. 11. For this particular Re , several computational results are available and they are compared with the results obtained with our CPU- and GPU-solvers. The present solvers captures fairly well the challenging initial square root singularity of the drag and is in agreement with the reference computation [24].

$Re = 9500$. This case is the highest Re for which simulations are presented in this work. Fig. 11 shows the evolution of the drag that we obtained with the CPU and the GPU solvers, in comparison to the reference solution. In Fig. 12 we observe the evolution of the vorticity at this Reynolds number. If we compare the drag obtained by the CPU solver with 1024×1024 particles (Fig. 3) with the one with the CPU solver employing 4096×1024 particles (Fig. 11) we note that there is a substantial difference. At $Re = 9500$ the emergence of the small scale is the driving mechanism in the dynamic of the flow, and therefore it is crucial to have a high resolution grid around the cylinder. In the case $Re = 1000$, a grid of 1024×1024 particles was

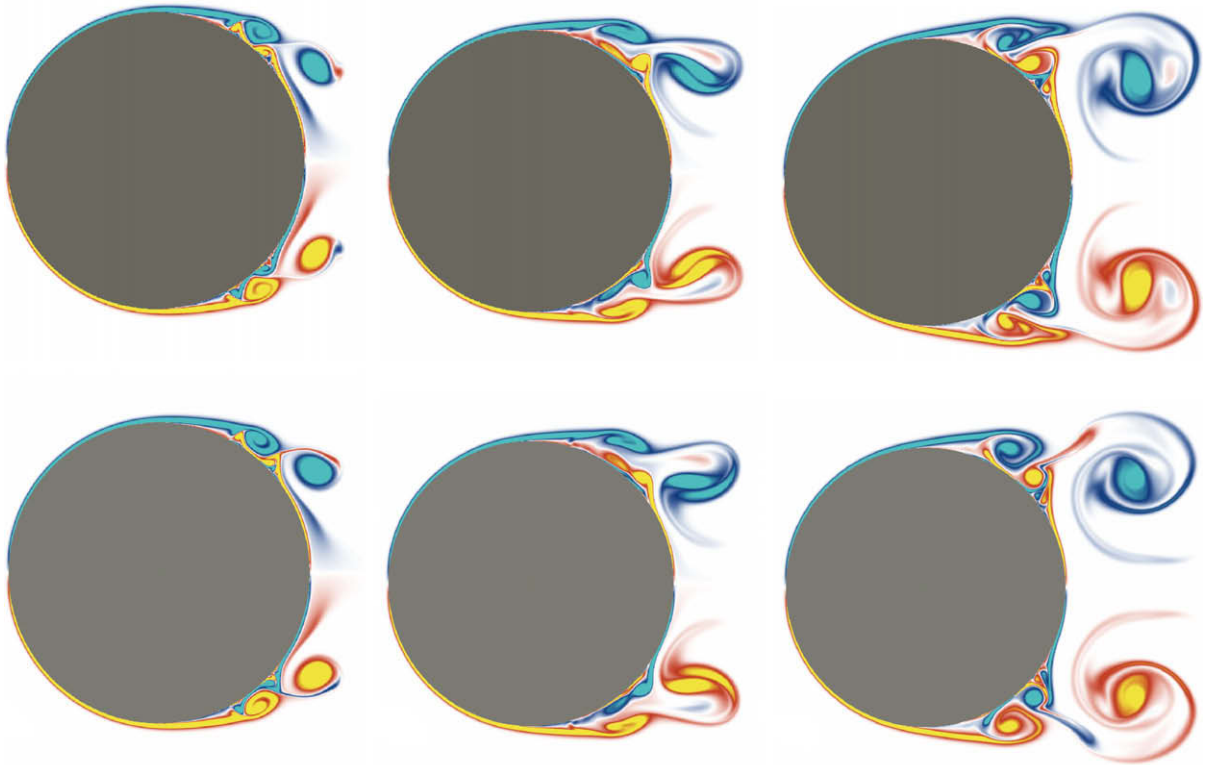


Fig. 12. Vorticity field obtained with the GPU solver (top) and CPU double precision solver (bottom) at $Re = 9500$ at time $T = 2.5, 3.5, 4.75$ (from left to right).

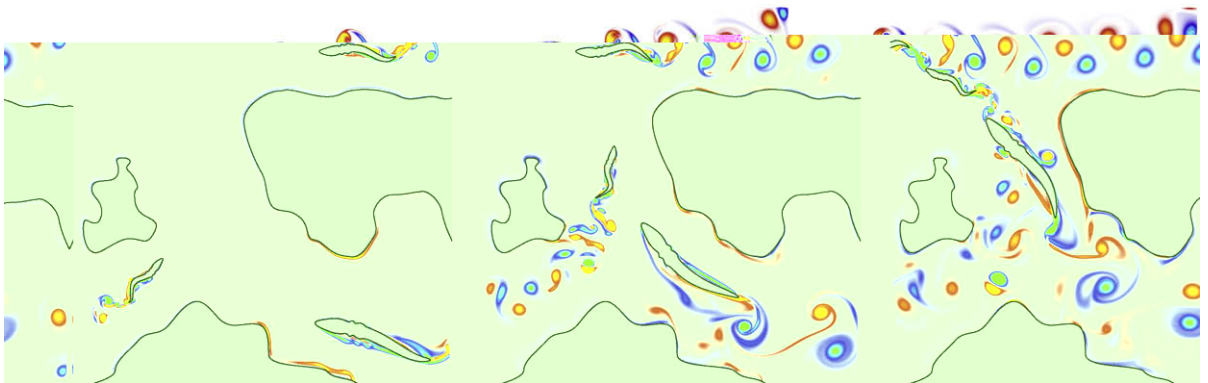


Fig. 13. Simulation of swimming entities in a complex geometry environment at different times (from left to right).

enough to capture the small scales, as it shown in Fig. 11. In the case of $Re = 9500$ this amount of particles is clearly not able to capture the small scale, as both CPU and GPU solvers fail to capture the correct drag/lift.

5.4. Handling complex and moving geometries

One benefit of the penalization technique is the capability of treating obstacles with complex geometry and obstacles that dynamically interact with the fluid (see [8]). In this work we demonstrate the capabilities of the method to handle simultaneously multiple deforming and stationary bodies by simulating anguilliform swimmers swimming in an environment with fixed obstacles. Snapshots of this simulations are illustrated in Fig. 13. The performance of the solver is the same as in the case of flow past a cylinder: fishes and background obstacles are drawn in two more cheap render-to-texture passes.

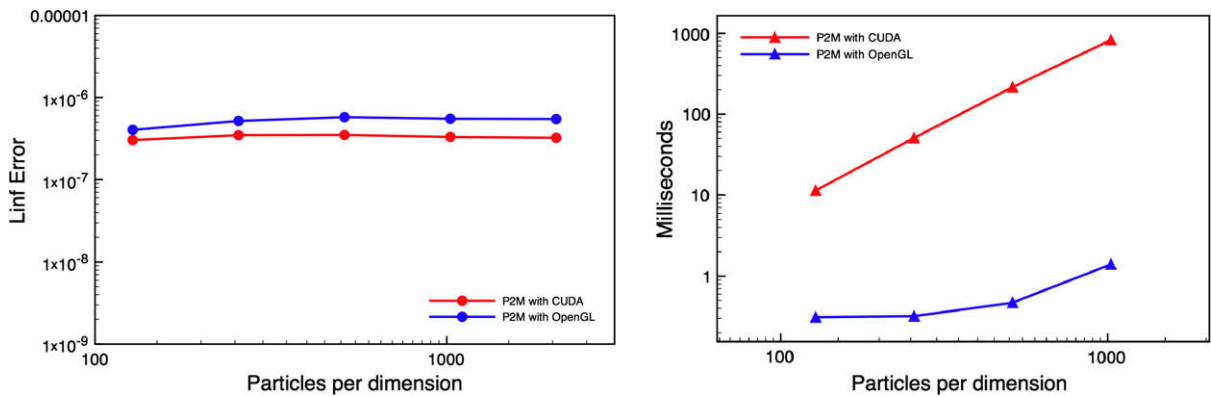


Fig. 14. Error (with respect to the same computation on the CPU in double precision) and computing time to perform a particle-mesh operation with CUDA (red) and OpenGL (blue). In the case of 1024×1024 particles OpenGL is 580 times faster than CUDA, which needs almost one second to carry out the operation. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

6. Performance

In this section we show the efficiency of performing particle-mesh operations using the framebuffer features of the GPU. We also report the overall performance improvement with respect to a single-core CPU solver and we analyze the distribution of time allocation for substeps of the method.

6.1. The use of OpenGL in particle-mesh operations

From the CUDA programming model it appears immediately clear that particle-mesh operations (see Eq. (19)) cannot be efficiently performed by CUDA-based solvers. The reason is that one is forced to use atomic operations and floating point atomic operations, which do not have hardware-support. This limitation is controversial since particle-mesh operations consist just of “blending” contributions from particles to grid points and therefore are extremely suitable operations to perform on the GPUs (in graphics). We thus developed particle-mesh operations in OpenGL, by treating every particle as a point-sprite and accumulating the contributions by blending the different point-sprites [39].

We show how much faster OpenGL is compared to CUDA in performing particle-mesh operations, with a comparable accuracy. We performed different tests by performing particle-mesh operations with particles displaced at random locations with random vorticity. As shown in Fig. 14, OpenGL can be up to 580 times faster than CUDA.

We consider that this is an interesting point for the GPGPU community: CUDA increases the productivity but at the same time hides some important hardware features. Thanks to the CUDA-OpenGL interoperability, one can exploit both productivity and access to all hardware functionalities. Another remark is that the OpenGL particle-mesh implementation is “compact”, for two reasons. Firstly it needs just one shader in one rendering pass, and not a combination of different shaders in different passes. Secondly, the shader is compact also in the number of lines: the vertex shader has 5 lines, the geometry shader has 7 lines while for the pixel shader 12. The straightforward CUDA implementation that we used to generate the plots in Fig. 14 consists of 137 lines. Improving its performance involves workarounds (substantially increasing the amount of code) to circumvent the parallel scattering of the particle-grid operations, probably by using some location-processing data structures.

6.2. Performance improvements

We compare the timings obtained with the GPU solver with respect to a reference CPU solver and a CPU-optimized solver for a flow past an impulsively started cylinder with 512×512 and 1024×1024 particles. The CPU taken in consideration is a 2.16 GHz Intel Core 2 Duo with 1 GB 667 MHz DDR2 SDRAM. The reference solver was written in C++, in single floating point arithmetic and was compiled with the Intel C++ compiler 10.0 with the `-O3 -axT` for a single-core execution and linked against the FFTW 3.0 libraries, configured with `--enable-sse2`. We then performed measurements on the test cases with the CPU reference solver using the Intel VTune profiler. Based on those measurements we wrote an optimized solver that extensively use SSE intrinsics, including the SSE3 instructions for horizontal adds. The OpenGL/CUDA solver was integrated in a set of C++ classes, using the Intel C++ 10.0 Compiler with `-O3`, although the impact of the optimization flags was practically negligible as C++ was used to “wrap” the different parts of the solver. Fig. 16 shows the strong speedup obtained with respect to the reference CPU solver. While the speedup of the CPU-optimized solver is roughly 1.4, the GPU solver has a speedup of almost 25.8. Fig. 17 shows how the computing time is divided into the different steps of the algorithm. From this

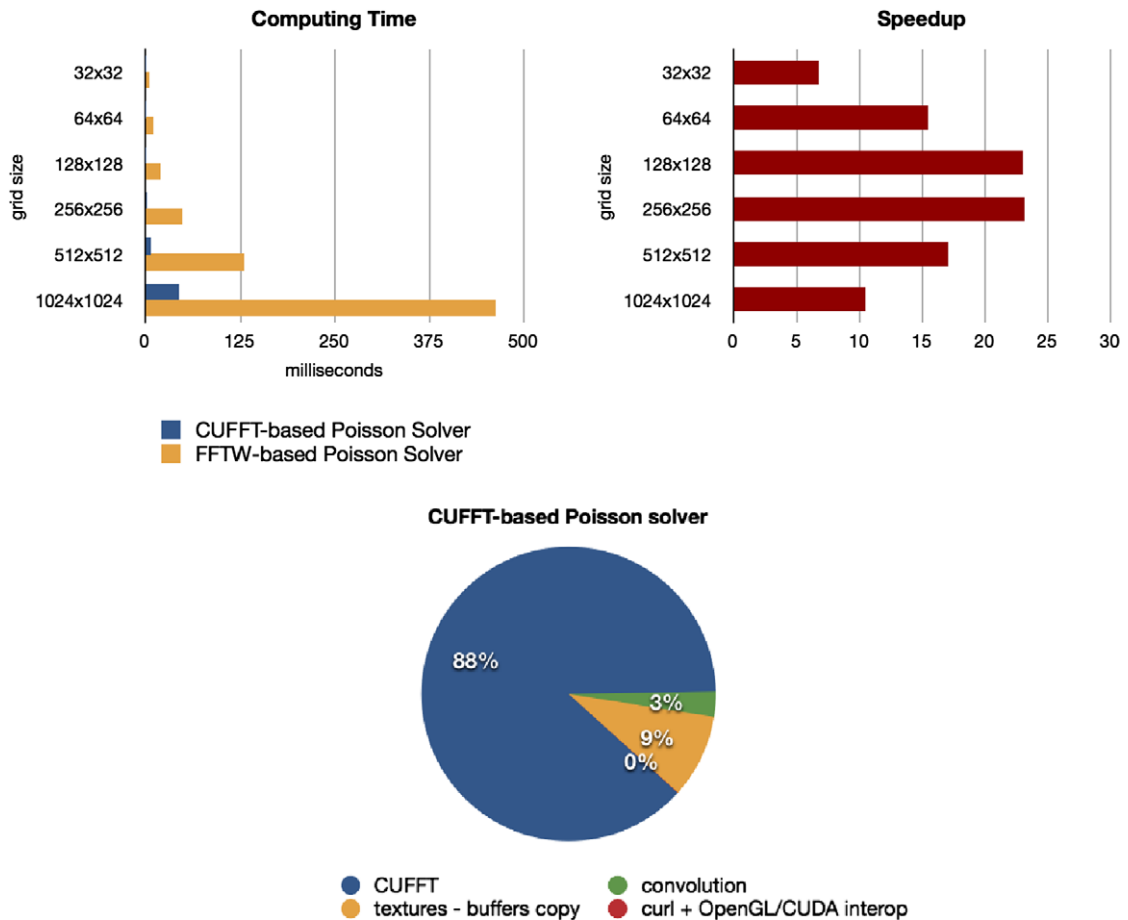


Fig. 15. Comparison of compute time (top-left) and speedup (top-right) of the CUFFT-based (GPU) Poisson solver over the FFTW-based (CPU) for different grid sizes. The most expensive part inside the CUFFT-based Poisson solver is the Fourier transform performed by CUFFT, which takes 88% of the time (bottom).

we observe that CPU and GPU have a completely different distribution, the GPU solver spends approximately 95% for solving the Poisson equation. As the Poisson solver is one of the key issue in the performance of this method, we provide a table of timings for different grid sizes in Fig. 15.

Within this 95% we also take into consideration the time spent to copy the vorticity in a CUDA-registered buffer (as a right-hand side) and take $\nabla \times \Psi$ from another CUDA-registered buffer. The redundant copy operations covers roughly 10% of the time spent solving the Poisson equation. The GPU solver relies on the CUDA/OpenGL interoperability. The CUDA part takes care about solving the Poisson Equation using CUFFT, whereas the integration steps and the particle-grid operations are performed within OpenGL. We recall that since we require unbounded boundary conditions, the size of the Poisson equation is 2048×2048 in the case of 1024×1024 particles and 1024×1024 in the case of 512×512 particles.

7. Discussion

The present algorithm exploits the features of remeshed particle methods, which employ particles in the advection steps and grids for the field equations. This allows for simulations using large time steps (far exceeding classical CFL conditions), fast computations of differential operators and accurate results. The efficient communication between particles and grid (particle-grid and grid-particle operations) is a crucial aspect of the algorithm while its simplicity make this method suitable for a GPU implementation.

This was confirmed by the performance we observed: the GPU version is almost 30 times faster than our CPU single-core solver. The GPU solver was able to simulate 22 time steps per second, using one million particles in an unbounded domain (and roughly 70 time steps per second in the case of periodic boundary conditions). Because CUDA (2.2) and OpenGL cannot share textures but only buffers, we had to maintain a redundant copy of the vorticity in a pixel buffer object and send that to the CUDA environment.

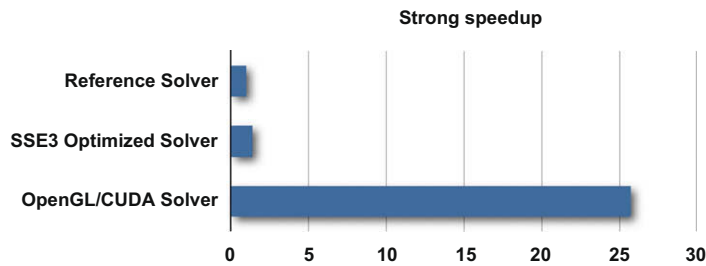


Fig. 16. Strong speedup (with respect to the reference CPU solver) of the CPU-optimized solver and the GPU solver.

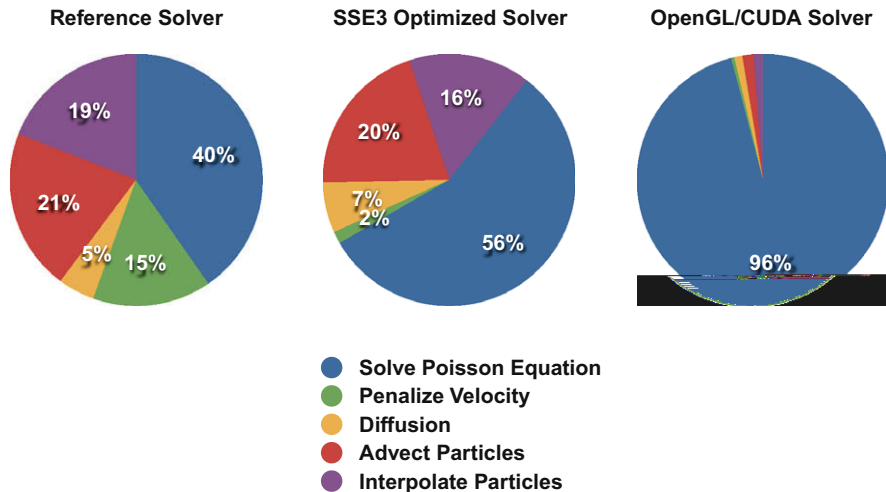


Fig. 17. Computing time allocation of the different steps of the algorithm for the reference solver (left), CPU-optimized solver (center) and GPU solver (right).

A key aspect in the performance of the solver (and intrinsic to remeshed particle methods) was the efficient implementation of particle-mesh operations. We show that through OpenGL we can perform such operations very efficiently and almost 3 orders of magnitude faster than CUDA. As the OpenGL longevity is at least as big as the one of GPGPU, we believe that performing particle-mesh operations in OpenGL will remain a viable solution also in future. We therefore observe that the solver works only because of the OpenGL/CUDA interoperability, which allows some communications through shared buffers.

Apart from performance, there are some differences between the accuracy of the GPU implementation and the CPU version. We observed several reasons which lead to this discrepancies, which are present in all the substeps of the method. The first reason is about the different ordering of the floating point instruction on the GPU with respect to the CPU. The second reason is about the different floating point arithmetics: CPU solver performed the floating point operations on the 80-bits coprocessor, leading to a superior accuracy with respect to a plain 32-bit floating point processor. The GPU models used to obtain the presented results, instead, are working with a special floating point arithmetic that does not support denormalized numbers and does not provide the standard rounding functions. The third reason is the maximum attainable sizes. They are different for GPU and CPU: while the CPU solver could tackle systems with 4096×4096 elements, the GPU solver is constrained to system sizes of 1024×1024 particles; although OpenGL supports big system sizes up to 8192×8192 elements, the CUFFT library on the employed GPUs could not run FFTs bigger than 2048×2048 (even if in theory it supports transform up to sizes of 16384).

The simulations performed using the presented algorithm were able to capture the drag/lift coefficients in the case of incompressible flows past cylinders, for both the impulsively start phase and the vortex shedding phase, from $Re = 40$ to $Re = 9500$. The efficiency of the present simulations and the accuracy in which they capture the vorticity field offer new possibilities for investigating vortex based control in bluff body flows [21]. Since the penalization technique is based solely on the characteristic function χ_s , it naturally supports obstacles with complex geometries. It is especially well suited for cases where the obstacle is represented implicitly by a level set function [8].

8. Conclusions

We report the GPU implementation of a remeshed vortex particle method along with the Brinkman penalization technique for simulations of bluff body flows. The method has been validated in simulations of flows past an impulsively started cylinder and it is shown to be second order accurate in time and space over a range of Reynolds numbers. The simplicity of the method, along with fast particle-mesh interpolations using OpenGL shaders, the floating point blending and the use of CUFFT for the solution of the Poisson equation allows for an efficient GPU implementation using one million particles at 21 steps/frames per second for bluff body flows in an unbounded domain. The present GPU implementations relies on the interoperability between CUDA and OpenGL. Current GPU architectures have limitations in the single floating point arithmetic, in the memory access on the GPU and the high degree of parallelism required in the algorithm in order to achieve good performance. We find however that the GPUs allow up to 30 times speed up of CPU-computations without significant deterioration (less than 5%) of computed quantities such as the vorticity field and the drag coefficient over a wide range of Re numbers. Ongoing work aims to exploit the speed of this computational tool and allow for the simulation of different scenarios of interacting bluff bodies such as swimming swarms.

Acknowledgements

We wish to acknowledge several helpful discussions with Dr. Massimiliano Fatica (NVIDIA) throughout the course of this work.

References

- [1] P. Angot, C.H. Bruneau, P. Fabrie, A penalization method to take into account obstacles in incompressible viscous flows, *Numerische Mathematik* 81 (4) (1999) 497–520.
- [2] E. Arquis, J.-P. Caltagirone, Sur les conditions hydrodynamique au voisinage d'une interface milieu fluide-milieu poreux: application à la convection naturelle, *Comptes Rendus de l'Academie des Sciences Paris* 299 (Série II) (1984) 1–4.
- [3] M. Bergdorf, G.-H. Cottet, P. Koumoutsakos, Multilevel adaptive particle methods for convection–diffusion equations, *Multiscale Modelling and Simulation* 4 (1) (2005) 328–357.
- [4] M. Bergdorf, P. Koumoutsakos, A Lagrangian particle-wavelet method, *Multiscale Modeling and Simulation* 5 (3) (2006) 980–995.
- [5] H. Brinkman, A calculation of the viscous force exerted by a flowing fluid on a dense swarm of particles, *Applied Scientific Research Section A-Mechanics Heat Chemical Engineering Mathematical DS*, 1947, pp. 27–34.
- [6] P. Chatelain, A. Curioni, M. Bergdorf, D. Rossinelli, W. Andreoni, P. Koumoutsakos, Billion vortex particle direct numerical simulation of aircraft wakes, *Computer Methods in Applied Mechanics and Engineering* 197 (13–16) (2008) 1296–1304.
- [7] P. Chatelain, A. Curioni, M. Bergdorf, D. Rossinelli, W. Andreoni, P. Koumoutsakos, *Vortex Methods for Massively Parallel Computer Architectures*, Springer-Verlag, Berlin, Heidelberg, 2008.
- [8] M. Coquerelle, G.H. Cottet, A vortex level set method for the two-way coupling of an incompressible fluid with colliding rigid bodies, *Journal of Computational Physics* (2008).
- [9] G.-H. Cottet, P. Koumoutsakos, M.L. Ould-Salihi, Vortex methods with spatially varying cores, *Journal of Computational Physics* 162 (2000) 164–185.
- [10] G.H. Cottet, P.A. Raviart, Particle methods for the one-dimensional Vlasov–Poisson equations, *Siam Journal on Numerical Analysis* 21 (1) (1984) 52–76.
- [11] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, *Journal of Computational Physics* 227 (24) (2008) 10148–10161.
- [12] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, High performance discrete fourier transforms on graphics processors, 2008 SC – International Conference for High Performance Computing, Networking, Storage and Analysis, 2008.
- [13] N.A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, *Journal of Computational Physics* 227 (18) (2008) 8290–8313.
- [14] M.J. Harris, Fast fluid dynamics simulation on the GPU, in: R. Fernando (Ed.), *GPU Gems*, vol. 38, Addison Wesley, 2004, pp. 637–665.
- [15] R. Henderson, Details of the drag curve near the onset of vortex shedding, *Physics of Fluids* 7 (9) (1995) 2102–2104.
- [16] R.W. Hockney, J.W. Eastwood, *Computer Simulation Using Particles*, second ed., Institute of Physics Publishing, Bristol, PA, USA, 1988.
- [17] T.Y. Hou, J. Lowengrub, Convergence of the point vortex method for the 3-d euler equations, *Communications on Pure and Applied Mathematics* 43 (8) (1990) 965–981.
- [18] D. Juba, A. Varshney, Parallel, stochastic measurement of molecular surface area, *Journal of Molecular Graphics and Modelling* 27 (1) (2008) 82–87.
- [19] G.H. Keetels, U. D'Ortona, W. Kramer, H.J.H. Clercx, K. Schneider, G.J.F. van Heijst, Fourier spectral and wavelet solvers for the incompressible Navier–Stokes equations with volume-penalization: convergence of a dipole-wall collision, *Journal of Computational Physics* 227 (2) (2007) 919–945.
- [20] N.K.-R. Kevlahan, O.V. Vasilyev, Ad adaptive wavelet collocation method for fluid–structure interactions at high reynolds numbers, *SIAM Journal on Scientific Computing* (2005).
- [21] P. Koumoutsakos, Active control of vortex–wall interactions, *Physics of Fluids* 9 (12) (1997) 3808–3816.
- [22] P. Koumoutsakos, Inviscid axisymmetrization of an elliptical vortex, *Journal of Computational Physics* 138 (2) (1997) 821–857.
- [23] P. Koumoutsakos, Multiscale flow simulations using particles, *Annual Review of Fluid Mechanics* 37 (1) (2005) 457–487.
- [24] P. Koumoutsakos, A. Leonard, High-resolution simulations of the flow around an impulsively started cylinder using vortex methods, *Journal of Fluid Mechanics* 296 (1995) 1–38.
- [25] P. Koumoutsakos, D. Shiels, Simulations of the viscous flow normal to an impulsively started and uniformly accelerated flat plate, *Journal of Fluid Mechanics* 328 (1996) 177–227.
- [26] A. Kravchenko, P. Moin, K. Shariff, B-spline method and zonal grids for simulations of complex turbulent flows, *Journal of Computational Physics* 151 (2) (1999) 757–789.
- [27] L. Lee, R.J. Leveque, An immersed interface method for incompressible Navier–Stokes equations, *SIAM Journal on Scientific Computing* 25 (3) (2003) 832–856.
- [28] W. Li, X.M. Wei, A. Kaufman, Implementing lattice Boltzmann computation on graphics hardware, *Visual Computer* 19 (7–8) (2003) 444–456.
- [29] M.N. Linnick, H.F. Fasel, A high-order immersed interface method for simulating unsteady incompressible flows on irregular domains, *Journal of Computational Physics* 204 (2005) 157–192.
- [30] Y. Liu, X. Liu, E. Wui, Real-time 3d fluid simulation on GPU with complex obstacles, 2004.
- [31] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, J.S. Vetter, Accuracy and performance of graphics processors: a quantum monte carlo application case study, *Parallel Computing* 35 (3) (2009) 151–163.
- [32] R. Mittal, G. Iaccarino, *Immersed Boundary Methods*, 2005. *Annual Review of Fluid Mechanics*.
- [33] NVIDIA, *CUDA CUFFT Library Manual*, June 2009.

- [34] M.L. Ould-Salihi, G.H. Cottet, M. El Hamraoui, Blending finite-difference and vortex methods for incompressible flow computations, *Siam Journal on Scientific Computing* 22 (5) (2001) 1655–1674.
- [35] C.S. Peskin, Numerical-analysis of blood-flow in heart, *Journal Of Computational Physics* 25 (3) (1977) 220–252.
- [36] P. Poncet, Methodes particulaires pour la simulation des sillages tridimensionnels, Technical Report, Universite de Grenoble I - Joseph Fourier, 2001.
- [37] P. Poncet, Topological aspects of three-dimensional wakes behind rotary oscillating cylinders, *Journal of Fluid Mechanics* 517 (2004) 27–53.
- [38] P.A. Raviart, An analysis of particle methods, *Lecture Notes in Mathematics* 1127 (1985) 243–324.
- [39] D. Rossinelli, P. Koumoutsakos, Vortex methods for incompressible flow simulations on the GPU. *Visual Computer* 24(7–9) (2008) 699–708. 26th International Conference on Computer Graphics, Istanbul, Turkey, Jun 09–11, 2008.
- [40] J. Stam, *Stable Fluids*, 1999.
- [41] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry* 28 (16) (2007) 2618–2640.
- [42] D. Tritton, Experiments on the flow past a circular cylinder at low reynolds numbers, *Journal of Fluid Mechanics* 6 (4) (1959). 547–.
- [43] O.V. Vasilyev, N.K.-R. Kevlahan, Hybrid wavelet collocation-Brinkman penalization method for complex geometry flows, *International Journal for Numerical Methods in Fluids* 40 (2002) 531–538.
- [44] X.M. Wei, Y. Zhao, Z. Fan, W. Li, F. Qiu, S. Yoakum-Stover, A.E. Kaufman, Lattice-based flow field modeling, *IEEE Transactions on Visualization and Computer Graphics* 10 (6) (2004) 719–729.