Visual Comput (2008) 24: 699–708
DOI 10.1007/s00371-008-0250-z

ORIGINAL ARTICLE

Diego Rossinelli
Petros Koumoutsakos

Vortex methods for incompressible flow simulations on the GPU

Published online: 17 May 2008
© Springer-Verlag 2008

D. Rossinelli (✉) · P. Koumoutsakos
Chair of Computational Science, ETH
Zurich CH-8092, Switzerland
diegor@inf.ethz.ch, petros@ethz.ch

Abstract We present a remeshed vortex particle method for incompressible flow simulations on GPUs. The particles are convected in a Lagrangian frame and are periodically reinitialized on a regular grid. The grid is used in addition to solve for the velocity–vorticity Poisson equation and for the computation of the diffusion operators. In the present GPU implementation of particle methods, the remeshing and the solution of the Poisson equation rely on fast and efficient mesh-particle interpolations. We demonstrate that particle remeshing introduces minimal artificial dissipation, it enables a faster computation of differential operators on particles over grid-free techniques and it can be efficiently implemented on GPUs. The results demonstrate that, contrary to common practice in particle simulations, it is necessary to remesh the (vortex) particle locations in order to solve accurately the equations they discretize, without compromising the speed of the method. The present method leads to simulations of incompressible vortical flows on GPUs with unprecedented accuracy and efficiency.

Keywords Vortex methods · Particles · Fluids · Graphics processors

1 Introduction

Flow simulations using particles have been used extensively in computer graphics (CG) [7, 8, 24] and in computational fluid dynamics (CFD) ([15] and references therein). These two lines of research have progressed independently and a methodological gap exists between particle flow simulations with requirements for visual realism in CG and particle simulations, with accurate flow physics in CFD. When we examine this gap we identify the loss of smooth particle overlap as the key source of inaccuracy in grid-free particle methods [5]. The goal of this paper is to bridge the gap by relaxing the grid-free character of particle methods, while maintaining their Lagrangian adaptivity, and by implementing remeshed vortex particle methods on GPUs for fast and accurate incompressible flow simulations.

1.1 Flow simulations on GPUs: grid free or accurate?

The use of GPUs has opened a new venue for fast simulations of fluids in the CG community. Flow simulations on GPUs using grid based methods have been first reported in [16] along with particle methods for visualization. More recent works [11, 23, 25] have focused on enhanced visual realism rather than numerical accuracy while grid based flow solvers with enhanced accuracy have been reported in [9, 22, 26]. In the grid based methods the discretization of the non-linear convection term is largely responsible for numerical dissipation. The discretization of the non-linear term is avoided by the Lagrangian formulation of the flow equations and their discretization using particle methods, such as smooth particle hydrodynamics (SPH) ([19] and references therein) and vortex methods (VM) ([5, 15] and references therein). SPH simulations have been used extensively for flow simulations in CG [20, 21] while more recently SPH techniques have been implemented in GPUs [1, 10, 12]. These flow simulations exhibit visual realism, but they fail to accurately solve the equations they discretize. This fact has been largely overlooked in the CG community while in CFD SPH and vortex techniques have been broadly recognized in the past as modeling tools with unknown levels of uncertainty in their numerical accuracy.

Inrecent years, starting from work in vortex methods [14] and their extension to SPH [4], it has been shown that regularization of the particle locations is necessary in order for particle methods to converge to the solution of the equations that have been discretized. The particle remeshing introduces artificial viscosity that can be reduced below the dissipation introduced by the discrete solution of the equations of motion. In simulations of incompressible fluids, SPH methodologies employ arbitrary pressure density relations (e.g. a linear pressure-density relationship in [10]) that do not correspond to realistic liquids and they result in incompatible sets of equations. In VM, incompressibility is implicitly satisfied by using the vorticity–velocity formulation, at the cost of solving a Poisson equation for deriving the velocity field. Furthermore, VMs offer additional savings of computational elements in cases where the vorticity field is limited in certain parts of the flows, as in the case of flows past bluff obstacles. At the same time we note that the remeshed particle locations can be used to discretize the diffusion of the particles resulting in one order of magnitude increase in computational speed over grid-free SPH methodologies. The remeshing and the need for solving a Poisson equation rely on effective grid-particle interpolations that requires their efficient implementation on the GPUs, due to data-scattering associated with the particle-mesh communications.

In this paper we present the effects of remeshing and non-remeshing particles in canonical fluid mechanics problems and we present the first implementation ever, to the best of our knowledge, of accurate vortex particle methods in GPUs.

2 Vortex methods

Vortex methods discretize the Navier–Stokes equations of an incompressible, viscous flow in a velocity–vorticity $(\boldsymbol{u}, \omega = \nabla \times \boldsymbol{u})$ formulation which in 2D is expressed in a Lagrangian form as:

$$\frac{D\omega}{Dt} = \nu \Delta \omega, \tag{1}$$

where $\frac{D}{Dt}$ denotes the material derivative and ν the viscosity of the flow. The velocity field $\boldsymbol{u}$ is obtained by solving the Poisson equation

$$\Delta \boldsymbol{u} = -\nabla \times \omega \tag{2}$$

with suitable boundary conditions [5]. We can discretize the vorticity field with a set of particles that have positions $\{\boldsymbol{x}_p\}$ and carry vorticity $\{\omega_p\}$ as:

$$\omega^h_\epsilon(x) = \sum_p \omega_p h^d \zeta_\epsilon\big(\boldsymbol{x} - \boldsymbol{x}^h_p\big), \tag{3}$$

where h denotes the interparticle distance and ζ_ϵ is a kernel with mollification/particle size ϵ. This approximation implies smoothing and discretization of the vorticity flow field with associated errors of $\mathcal{O}(\epsilon^r)$ and $\mathcal{O}((\frac{h}{\epsilon})^m)$ respectively [5]. The factor r depends on the moments of ζ_ϵ while m can be very large and depends on the quadrature rule. This result implies that smooth particles must overlap, or in other words that the interparticle distance must be smaller than the mollification kernel. This requirement cannot be satisfied in general as particles deform according to the velocity of the flow field. From Eq. 1 we can derive the time evolution for the location and strength of each computational element:

$$\begin{cases} \dot{\boldsymbol{x}}_p = \boldsymbol{u}^h_\epsilon(\boldsymbol{x}_p) \\ \dot{\omega}_p = \nu \Delta \omega^h_\epsilon(\boldsymbol{x}_p). \end{cases} \tag{4}$$

The velocity can be recovered from the vorticity field by solving Eq. 2 on a grid. In this work we also take advantage of the presence of a grid to solve the diffusion equation when particles have been remeshed on the grid using a central finite difference scheme. Note that this differentiation accelerates 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.

On a given set of particles $\{(\omega_p, \boldsymbol{x}_p)\}$, the particle-mesh interpolation operation (denoted as $\boldsymbol{I}^M_P$) maps the particle vorticity onto grid nodes with grid spacing h as

$$\omega^{\text{mesh}}_m = \sum_p \omega_p \cdot W\left(\frac{1}{h}\big(\boldsymbol{x}^{\text{mesh}}_m - \boldsymbol{x}_p\big)\right), \tag{5}$$

where W is an interpolation kernel function. Given the vorticity on the mesh one can recover the vorticity of each particle by defining the mesh-particle operation $\boldsymbol{I}^P_M$:

$$\omega_p = \sum_m \omega^{\text{mesh}}_i \cdot W\left(\frac{1}{h}\big(\boldsymbol{x}_p - \boldsymbol{x}^{\text{mesh}}_m\big)\right). \tag{6}$$

The interpolation kernel can be expressed as a tensorial product $W(x, y) = W(x)W(y)$. In the present study the M'_4 kernel is used.

$$M'_4(x) = \begin{cases} 0 & \text{if } |x| > 2 \\ \frac{1}{2}(2-|x|)^2(1-|x|) & \text{if } 1 \le |x| \le 2 \\ 1 - \frac{5}{2}x^2 + \frac{3}{2}|x|^3 & \text{if } 1 \ge |x| \end{cases} \tag{7}$$

This M_4' interpolation kernel is equivalent to the Catmull–Rom spline interpolation used in CG.

2.1 Remeshing

Particle methods, when applied to the Lagrangian formulation of the convection–diffusion equation, enjoy automatic adaptivity of the computational elements as dictated by the flow map. This adaptation comes at the expense of the regularity of the particle distribution because 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 precisely 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 to inaccurate approximations of the related derivative operators. To remedy this situation, location processing techniques reinitialize the distorted particle field onto a regularized set of particles and simultaneously accurately transport the particle quantities. The accuracy of remeshing has been thoroughly investigated in [14] and it was shown to introduce numerical dissipation that is well below the dissipation introduced by other temporal and spatial discretizations. One way to regularize the particles is setting the new particle positions to be on the grid node positions and recomputing the transported quantities with a particle-mesh operation.

The particle-mesh operation ($\boldsymbol{I}_P^M$) on a particle set $S = \{(\omega_p, \boldsymbol{x}_p)\}$ with an interpolation kernel $W(.)$ is

$$\text{Remeshing}(\{(\omega_p, \boldsymbol{x}_p)\}) = \left(\boldsymbol{I}_P^M\{(\omega_p, \boldsymbol{x}_p)\}, \left\{\boldsymbol{x}_m^{\text{mesh}}\right\}\right), \quad (8)$$

with $\boldsymbol{I}_P^M$ defined by Eq. 5.

2.2 Time integration

Particle locations and weights are updated in order to satisfy in-turn convection and diffusion. For simplicity we introduce the algorithm in the context of a first order time integration scheme. Starting from particles on the grid nodes $S^n = \{(\omega_p^n, \boldsymbol{x}_p^{\text{mesh}})\}$ at time t, we now compute the next solution S^{n+1}, whose particle locations are also coinciding with the grid nodes at time $t+\delta t$. The first order time integration proceeds as follows:

1. Update the transported vorticity of the particles:
$$\omega_p^{n+1} := \omega_p^n + \delta t \nu \sum_{j\in\langle p\rangle} a_{j,p}^{\text{diffusion}} \omega_j^n .$$
2. Compute the right-hand side of the Poisson equation. With particles on the mesh a finite difference stencil is used:
$$-\nabla \times \omega^h\left(\boldsymbol{x}_p^{\text{mesh}}\right) = -\sum_{j\in\langle p\rangle} a_{j,p}^{\text{curl}} \omega_j^n .$$
3. Solve Eq. 2 on the mesh, for $\boldsymbol{u}(\boldsymbol{x}_p^{\text{mesh}})$.
4. Move the particles:
$$\boldsymbol{x}_p^{n+1} := \boldsymbol{x}_p^{\text{mesh}} + \delta t \boldsymbol{u}\left(\boldsymbol{x}_p^{\text{mesh}}\right).$$
5. Remesh to obtain regularized particles:
$$S^{n+1} = \text{Remeshing}\left(\left\{\left(\omega_p^{n+1}, \boldsymbol{x}_p^{n+1}\right)\right\}\right)$$

where $a_{j,p}^{\text{curl}}$, $a_{j,p}^{\text{diffusion}}$ are finite difference stencils approximating respectively the curl and Laplace operators; $\langle p\rangle$ denotes the set of neighbor particles of p.

We note here two important facts: this algorithm is fast because every differential operation is performed on the remeshed particle locations, without evaluating any kernel. Furthermore, as particles do not obey a classical CFL condition[1] on $\boldsymbol{u}$, we can take large time steps bounded by $\delta t \sim 1/\|\nabla \boldsymbol{u}\|_2$.

In the present work it is shown that higher order time integration schemes are necessary in order to allow for larger time steps and higher accuracy. For higher order time integration schemes, starting from the particles S^n, we have to evaluate the right-hand side in Eq. 4 multiple times and store the results into temporary variables. A basic difference besides the introduction of additional memory required for the right-hand sides, is that this algorithm also involves particle-mesh and mesh-particle operations outside the remeshing step: at every sub-step we have to map the particles onto a grid.

Note also that it is not required to remesh the particles after every step, but it is essential to perform remeshing steps periodically, e.g. after every n steps.

3 GPU implementation

In GPGPU the computational elements are often mapped to textures. In the present work the computational elements are both regularly spaced grid nodes and particles at arbitrary locations. Similar to [13] and [12], we employ an RGB texture to represent a set of particles where each texel contains the state of one particle. For two dimensional flows the red and green channel of a given texel represent the particle position, whereas the B channel indicates the transported vorticity. A one-to-one mapping between texels and grid nodes is used to represent the computational mesh with a texture.

3.1 Solver overview

The main workflow of our GPU-solver is illustrated in Fig. 1. The core components of the solver are shown in blue, the gray component identifies a tool used as a "black box", whose subcomponents are not further explored.

[1] Courant–Friedrichs–Lewy condition for hyperbolic partial differential equations.

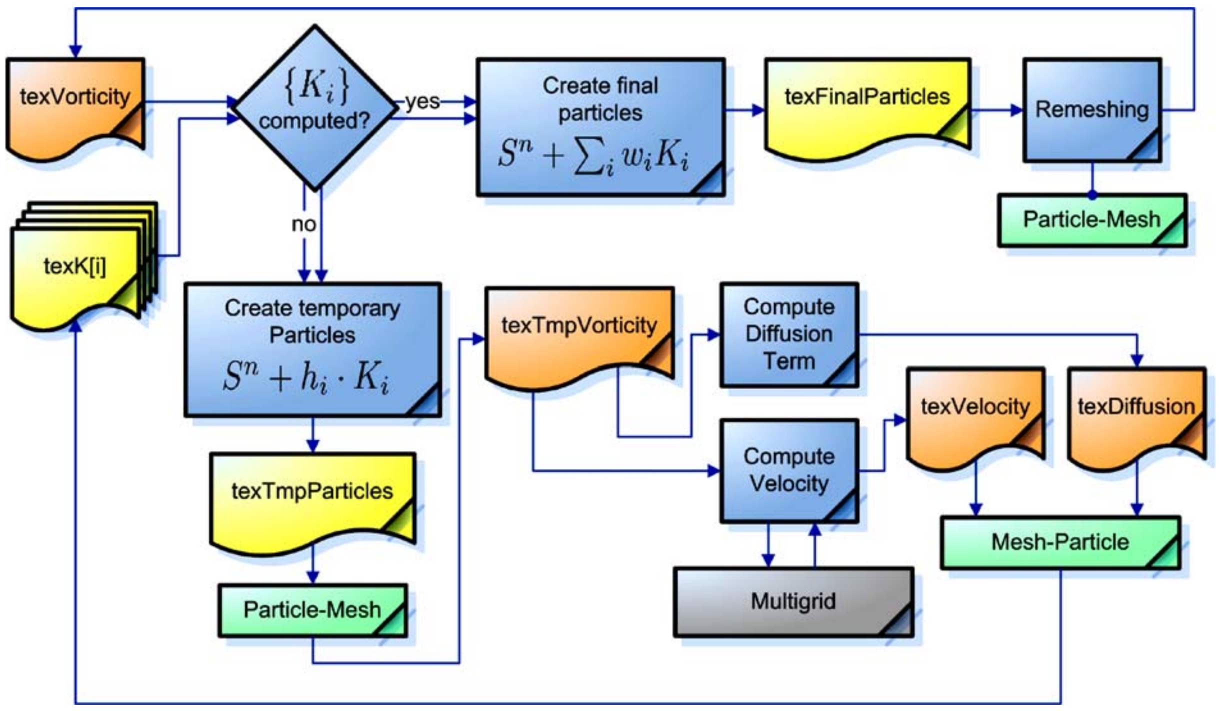


Fig. 1. The components of the GPU-solver with k-th order Runge–Kutta time integration

Each core component takes as input a set of textures and produces as output another set of textures. The managed data is represented with texture and is painted in orange or yellow. The yellow color signifies that the texture represents a particle set, therefore the RGB channels are (x_p, y_p, ω_p). The orange color indicates that the texture represents a grid, therefore it contains only ω. The green box represents the particle-mesh operation. Additionally to the texture `texTmpParticles`, the green box requires a vertex array of the same size.

The **particle-mesh** operation is performed using a tensorial product of one-dimensional interpolating kernels. The contributions of all particles on each grid node are computed iteratively. The use of interpolation kernels with compact support implies that the contributing particles are located close to the grid nodes. For each particle we can locate which grid nodes are affected and therefore add its respective contributions to these nodes. Based on this observation we present a concise method to perform data-scattering particle-mesh operations on the GPU.

Alternatives to this method would avoid direct data-scattering by handling more complex data structures used for location processing, as presented in [17]. One key characteristic of the proposed method, similar to [12], is to employ *point-sprite*[2] primitives, which allow the use of points rather than quads and are able to generate texture coordinates which are interpolated across the point. We start by having the status of the particle set stored in a texture `texParticles` and a vertex array with the size of `texParticles`. The output will be stored in the texture `texMesh`. The algorithm has the following steps:

1. Set the point size equal to the support of $W(\cdot)$.
2. Enable the *point-sprite* drawing mode.
3. Attach the `texMesh` as a render target and clear it with zeros.
4. Enable blending with 1 as source factor as well as destination factor.
5. Set the graphics pipeline as follows:
   Vertex shader:
   Read (x_p, y_p, ω_p) from `texParticles`. Store the transported vorticity ω_p as front color of the primitive and the location coordinates as the position of the vertex.
   Geometry shader:
   If a particle is close to the boundary, dynamically *clone* the particle to reproduce the right boundary conditions. If a particle has $\omega_p = 0$, discard the primitive.
   Fragment shader:
   Compute the vector distance $\boldsymbol{d} = (d_1, d_2)$ of the fragment with respect to the center of the point-sprite to produce *color* $= W(d_1)W(d_2)\omega_p$ as a result, where ω_p is the vorticity of the current point sprite.
6. Draw the vertex array as point-sprites.

As we are drawing point sprites, each vertex will be rasterized in a quad made of several fragments and different

[2] *www.opengl.org/registry/specs/ARB/point_sprite.txt*

texture coordinate values. The distance between the center of the point sprite and the generated fragments is known at the fragment stage, and it is stored as a texture coordinate. We re-scale appropriately the texture coordinate and we use it as an argument of $W(\cdot)$. Since we know the quantity carried by the particle (as it is stored as primitive color) we multiply these together to obtain the contribution of the particle to that grid node.

Enabling the blending mode, we can sum each contribution from every particle to any node and obtain as a result an interpolated grid from values transported by the particle set. For a given framebuffer (destination) pixel, the blending is performed as an atomic instruction. Therefore it cannot be performed in parallel with respect to the incoming source fragments. This could be a potential performance bottleneck. We can minimize however this problem by reducing the maximal number of incoming fragments per framebuffer pixel. This is possible if the particles are not concentrated on a particular region so that their contributions will be spread uniformly in the framebuffer. This is automatically guaranteed in the remeshing stage: since by remeshing we uniformly redistribute the particles in the domain, and thus avoid blending becoming a critical bottleneck.

Since the **mesh-particle** operation is essentially a data-gathering operation, it can be performed with a fragment shader, reading a texture representing the mesh and attaching the particle set texture as a render target. At the fragment stage we read a subset of mesh nodes by performing texture dependent texture-fetches, and we compute $\boldsymbol{I}_M^P$, eventually obtaining the carried quantity for each particle.

3.2 Solving the Poisson equation

In order to solve the Poisson equation for the velocity field $\boldsymbol{u}$ we developed a periodic 2D multigrid solver [3] for the GPU. The GPU-Multigrid is designed for cell-centered elements, and has prolongation and restriction of order 4. We validated the GPU-Multigrid against different test problems, and observed that on average (after 3–6 cycles), the relative residual was between 10^{-5} and 10^{-3} (in both L_2 and L_∞ norm). We noticed that, for the same physical domain, higher resolution discretization causes bigger residual ($\sim 10^{-4}$). The most probable reason is the single floating point precision limitation in the arithmetic.

We note that point-sprites are clipped automatically by the graphics pipeline and this could be a problem in the case of periodic boundary conditions, since contributions from some particles are discarded. In order to overcome this problem, we use a geometry shader to check if the kernel assigned to each particle "touches" the boundary. If this is the case, we create a new particle with the same vorticity and with a position translated by one domain length. In this way we can generate exactly the contributions that are discarded at the rasterization stage. The geometry shader is an elegant choice to solve this problem, however it is not the only one. We could perform a four-pass rendering with blending, where in each pass, a slightly shifted domain is considered and each particle has to be redrawn. This method gives exactly the same result as the geometry shader but it is much more expensive as every particle has to be rendered four times. Conversely, with the geometry shader, only the particles at the boundary have to be rendered twice (four times for the negligibly small particle set at the corners).

Furthermore, with the geometry shader we not only have the capability to create particles *on-demand*, but also to *discard* particles when they are unnecessary, i.e. when the transported vorticity is zero. This adaptivity additionally improves the performance of the proposed solver. Even if the particle-mesh operations are cheap when compared to other components of the solver, the performance difference between these two approaches is significant.

4 Results

The proposed particle solver has been validated on three benchmark flows. The GPU-solver was written in C++, using OpenGL, and each simulation was run on NVIDIA Quadro FX 5600.

4.1 The role of remeshing

In order to demonstrate the impact of the remeshing step, we consider the vorticity equation without the diffusion term ($\nu = 0$).

In this case the vorticity evolves according to the Euler equation $\frac{D\omega}{Dt} = 0$. As the initial condition we set a radial function:

$$\omega_0(\boldsymbol{x}) = W \cdot \max(0, 1 - \|\boldsymbol{x}\|/R), \tag{9}$$

where W is the maximum vorticity and R controls the support of ω_0. Since the vorticity is radially symmetric and there is no diffusion, the system is in a steady state: the exact solution in time is just the initial condition ($\omega(t) = \omega_0$). We can therefore use this problem as validation and study "the importance of remeshing" the particles during the simulation. Figure 2 shows the crucial difference between performing and not performing the remeshing step. In this case we used $W = 100$, $R = 0.5$, and a time step $\delta t = 5 \times 10^{-3}$. When no remeshing step is performed, the solver generates growing spurious structures which lead to a break in the radial symmetry of the vorticity field. This break causes a rapidly increasing inaccuracy of the computed solution.

4.2 Taylor–Green vortex

The Taylor–Green vortex problem represents the evolution of a complex vortical flow that admits an analytical

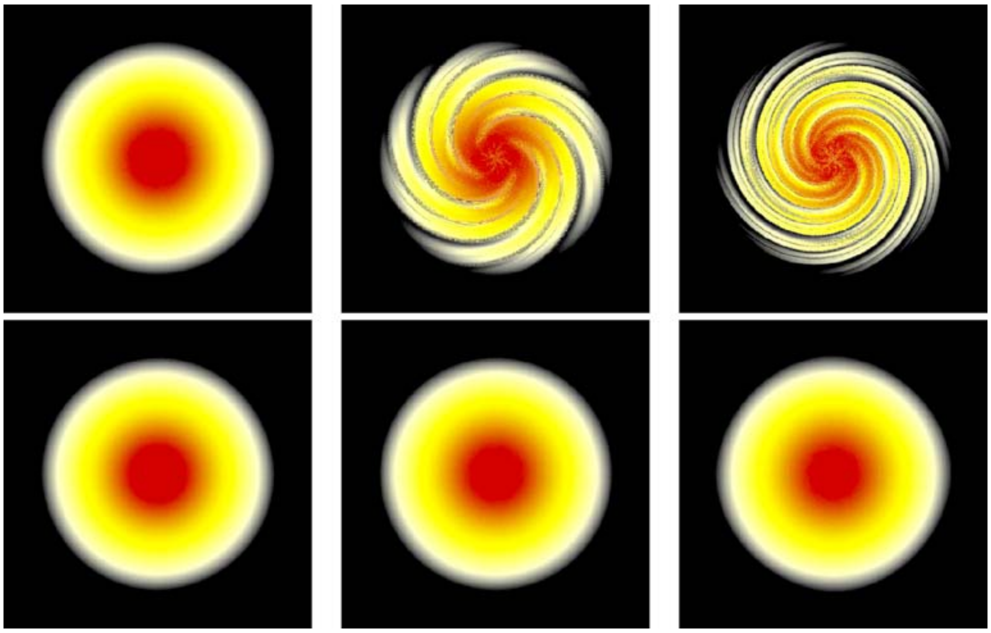


Fig. 2. The role of the remeshing step. Evolution of the vorticity at time $t = 0.01$, $t = 0.10$ and $t = 0.15$; using a second order time integrator for the Euler equation. The initial condition, which is a radial function with compact support, is already a steady state solution, and thus vorticity must not change in time. Without remeshing the simulation produces spurious artifacts that become progressively stronger (*top*). The remeshing step will prevent the creation of these artificial structures (*bottom*), if it is performed frequently enough (every five steps in this case)

solution. The vortices evolve in the unit square with periodic boundary conditions and the analytical solution for the velocity field is given by the following equations in nondimensional form:

$$\begin{cases} u(x, y, t) = -Ue^{bt}\cos(2\pi x)\sin(2\pi y) \\ v(x, y, t) = Ue^{bt}\sin(2\pi x)\cos(2\pi y). \end{cases} \tag{10}$$

Here, $b = \frac{-8\pi^2}{\mathrm{Re}}$ and the viscosity ν is $\frac{1}{\mathrm{Re}}$, where Re is the Reynolds number.

The problem has been tested in the past in the context of smooth particle hydrodynamics in [4] where it was shown that remeshing is essential for a particle method to capture the evolution of the vorticity field. We performed simulations using $U = 0.5$ and $\mathrm{Re} = 10^6$ for a total time of $t_{\mathrm{end}} = 5 \times 10^{-3}$ using a time step of $\delta t = 10^{-5}$. The convergence of the present method is depicted in Fig. 3. We observe that the method exhibits second order convergence in space. We note however that for high resolution simulations, the rate of reduction decreases. This behavior is attributed to the utilization of single precision arithmetic available for the GPU. The comparison of different time integration techniques indicates that high-order time integrators are preferable over the Euler method providing up to three orders of magnitude higher accuracy for the same step size. In the right plot of Fig. 3 we observe that the norm of the error has grown. In this case the time step was $\delta t = 2 \times 10^{-4}$ and $t_{\mathrm{end}} = 0.1$. This behaviour is expected as we have performed many steps ($\sim 10^4$) with our solver and the time step was large compared with the one used for the left plot in Fig. 3.

The advantage of Runge–Kutta methods over Euler is again evident, even though in this case the error reduction is limited to one order of magnitude.

4.3 Thin double shear layer

The thin double shear layer is a challenging benchmark for incompressible flow solvers. Brown and Minion [18] have demonstrated that in under-resolved simulations spurious vortices infiltrate the numerical solution in discretizations by various computational methods. We have used the double shear layer benchmark to study the effects of low-resolution simulations in vortex methods. The domain is the unit square with periodic boundary conditions with the initial condition for the velocity field $\boldsymbol{u} = (u, v)$ in the following non-dimensional form:

$$\begin{cases} u(x, y) = \tanh(\rho \cdot \min(y - 0.25, 0.75 - y)) \\ v(x, y) = \delta \cdot \sin(2\pi(x + 0.25)). \end{cases}$$

In the present simulations we set $\delta = 0.05$, $\rho = 80$ and a viscosity $\nu = 10^{-4}$.

All simulations were performed using the fourth order Runge–Kutta with a timestep $\delta t = 0.02$ and $t_{\mathrm{end}} = 1.0$. The numerical results are depicted in Fig. 4 in the form

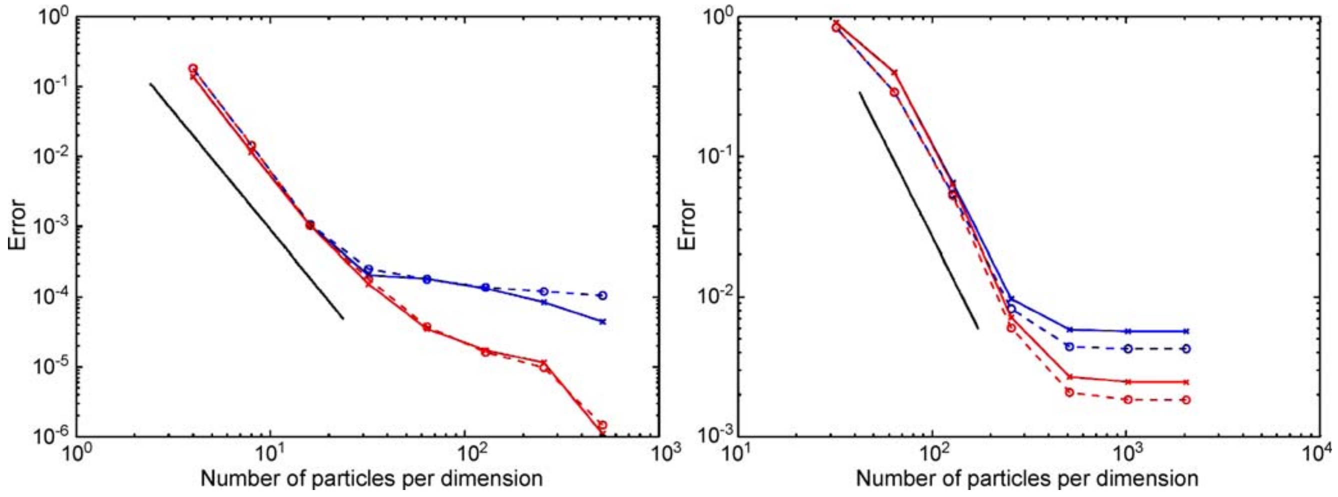


Fig. 3. L_2 errors (*contiguous lines*) and L_∞ errors (*dashed lines*) for the simulation of the Taylor–Green Vortex in terms of the number of computational elements. Euler time integration (*blue lines*) and fourth order Runge–Kutta (*red lines*). Settings: $U = 0.5$ and $\mathrm{Re} = 10^6$, final time of $t_{\text{end}} = 5 \times 10^{-3}$ (*left*) and $t_{\text{end}} = 0.1$ (*right*). *Black lines* denote a second order convergence

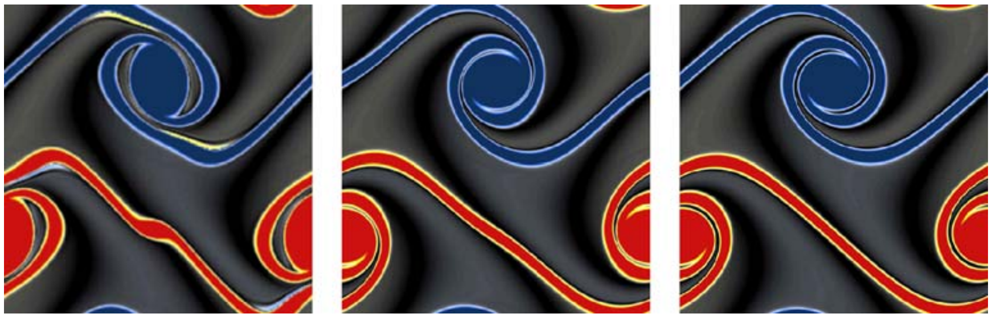


Fig. 4. Simulations of the thin double shear layer. *Red* denotes high positive vorticity, and *blue* denotes high negative vorticity. *Left* to *right*: simulation using 256×256, 512×512 and 2048×2048 particles. Note the development of a spurious vortex for the two lower resolutions

of vorticity for three different resolutions. The spurious vortices are eliminated by using 512×512 particles and an associated grid of 512×512 nodes. Note however that the solution shows some minor undulations instead of the expected straight line [18], near to the center of the domain. This numerical artifact completely disappears using 2048×2048 computational elements.

4.4 Random vorticity

Finally, we addressed a case of viscous vorticity decay from an initially uniform random distribution with an average of zero vorticity and a maximum value of 400. The domain is the unit square with periodic boundary conditions, and the viscosity was set to $\nu = 10^{-7}$. The first row of Fig. 5 shows the evolution of the flow obtained for a remeshed VM with a first order time integrator and a timestep $\delta t = 0.001$. The GPU-Multigrid configuration for the Poisson's equation consists of two V-Cycles with two Jacobi relaxation iterations at each level per timestep.

The second row shows the evolution of the same initial vorticity distribution when using the fourth order Runge–Kutta time integrator and a remeshed VM. The GPU-Multigrid configuration consists of four V-Cycles with four Jacobi relaxation iterations at each level per step. The timestep size was maintained at $\delta t = 0.001$. In the last row of pictures we show the vorticity field that we obtain without performing remeshing during the simulation. Note that the field does not develop the expected large scale structures but remains chaotic, a situation which may be perceived in fact as visually realistic but at the same time it represents a highly inaccurate solution of the Navier–Stokes equations. The utilization of a first order time integration scheme introduces a large amount of numerical viscosity producing large, weak vortex cores. On the other hand, the fourth order Runge–Kutta scheme succeeds in restraining the effects of

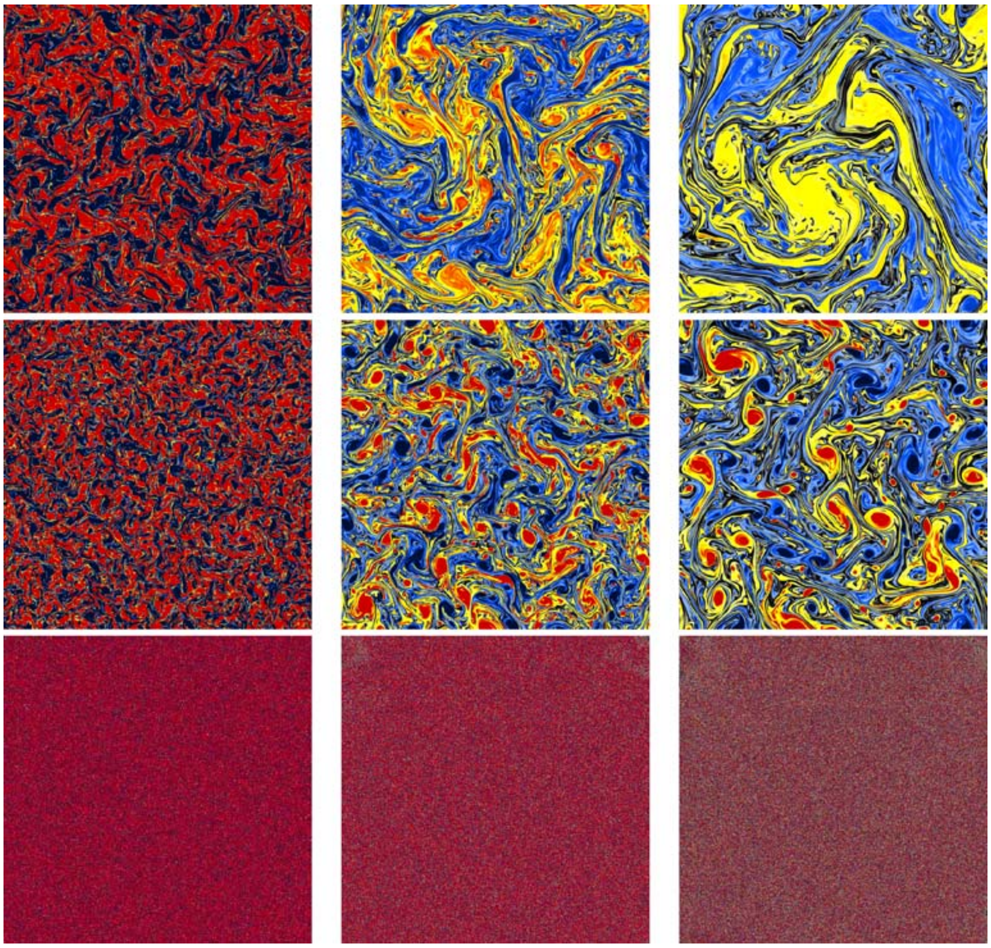


Fig. 5. Evolution of the vorticity starting from a uniform random distribution with maximum of 400 and average of zero, using a first order time integrator (*top*), a fourth order time integration (*middle*), a fourth order time integrator without remeshing during the simulation (*bottom*), at time $t = 0.01$, $t = 0.1$ and $t = 0.2$ (*left* to *right*)

numerical viscosity producing smaller vortices of higher intensity.

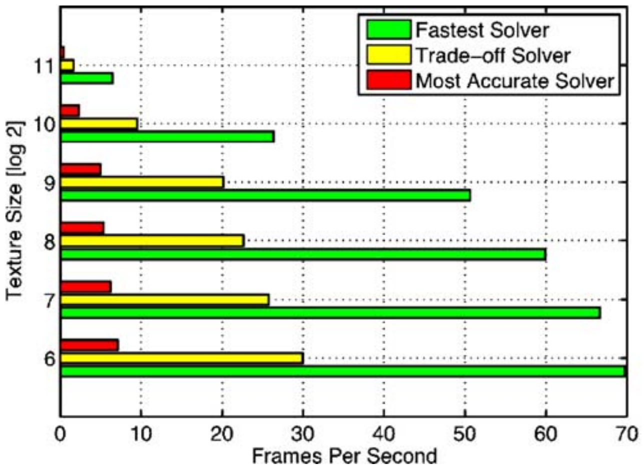
4.5 Performance

The presented solver involves a number of computational parameters, such as multigrid steps, order of time integration, etc. In order to quantify its performance we discuss three representative sets of parameters resulting respectively in: *Fastest*, *Trade-Off* and *Most Accurate* simulations. The *Fastest* set of parameters consists of a first order time integration, two V-Cycles with two Jacobi relaxation iterations at each level, per time step; the *Trade-Off* consists of a second order Runge–Kutta with two V-Cycles (four Jacobi relaxation iterations per level) per step. The last set of parameters corresponds to the one referred into Fig. 5 (fourth order time integration, four V-Cycles with four Jacobi relaxation iterations at each level, per timestep). As indicated in Fig. 6, one can achieve more than 25 FPS using a set of 1024×1024 particles with the *Fastest* set of parameters. The *Trade-off* configuration barely achieves ten FPS with the same number of particles. The main decrease in performance is noticed by passing from 1024×1024 to 2048×2048 particles. For the *Most Accurate* configuration we observe the least change in performance, revealing that the texture size is not the most performance-critical parameter in this case.

As we have mentioned, the remeshing step can be performed either with a multi-pass approach or by utilizing a geometry shader. The diagram on the right of Fig. 6 summarizes the performance of solving the random vorticity problem for both approaches as a function of the utilized number of particles. It is obvious that the geometry shader approach always is the fastest, in particular when we use 1024×1024 particles, where we obtain a speed up of 1.5, as on average the geometry shader has to render each particle just once. The multi-pass approach, on the other hand, processes each particle four times (at least at the vertex stage).

5 Discussion and conclusions

We have implemented a vortex method for the accurate and efficient simulations of incompressible vortical flows



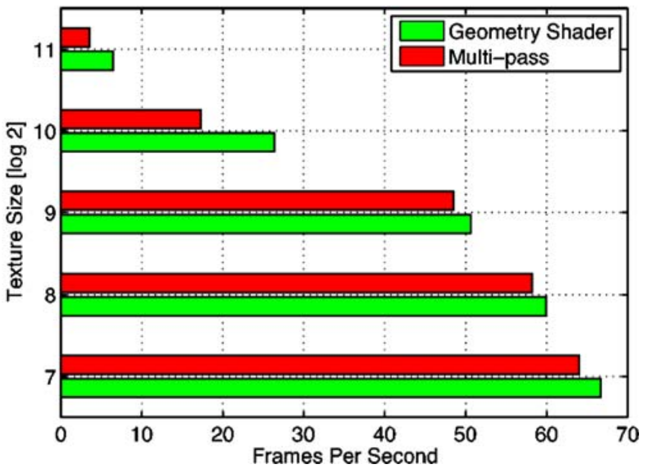


Fig. 6. Overall performance measurements: on the *left* we compare three different configurations of our solver: *Fastest* (Euler, with a rough GPU-multigrid), *Trade-Off* (second order Runge–Kutta, with an accurate GPU-multigrid but few cycles) and *most accurate* (fourth order Runge–Kutta, accurate GPU-multigrid). On the *right* we compare the performance of the remeshing by using a multi-pass rendering method and by using a geometry shader

on GPUs. The solver relies on the remeshing of the particle locations, it runs exclusively on the GPU and it is shown to have second order accuracy in space and up to fourth order accuracy in time.

These methods have been validated in challenging benchmark flow problems demonstrating for example that even with the limitation of single precision arithmetic, we are able to obtain a second order convergence in space in the case of the Taylor–Green vortex and to recover the correct solution for the evolution of a thin double shear layer. Furthermore we have shown the importance of adopting high-order time integration methods to achieve accuracy. In turn we have demonstrated that the performance of the GPU-solver depends critically on the set of computational parameters: The *fastest* set allows flow simulations with 1024×1024 particles at 25 FPS, whereas the *most accurate* only achieves three FPS, albeit with a significantly higher accuracy. The particle-mesh communication is critical for the present method and it is performed by using a geometry shader, texture fetch at the vertex stage and the floating point framebuffer/blending. The present method maintains the Lagrangian adaptivity of particle methods, it presents the first particle-mesh technique for flow simulations on the GPU and it allows for accurate, real time simulations of incompressible flows.

Present work focuses on extending the solver to 3D domains. One critical issue in this effort is associated with the efficiency of the method, since texture-fetches of 3D textures are not as fast as the 2D ones. In order to avoid this deterioration of performance, a possible strategy involves the use of representation of the 3D domain with a set of 2D textures while ensuring consistency (by clipping and particle replications) at the boundaries of each subdomain.

Future work will focus on the GPU-implementation of multiresolution vortex particle methods [2] and particle methods capable of handling effectively complex, deforming geometries [6]. We envision that such implementation will help materialize the real time simulation of challenging fluid mechanics phenomena with visual realism and accuracy of the flow physics.

References

1. Amada, T., Imura, M., Yasumuro, Y., Manabe, Y., Chihara, K.: Particle-based fluid simulation on GPU. In: ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH 2004. Los Angeles, CA (2004)
2. Bergdorf, M., Koumoutsakos, P.: A Lagrangian particle-wavelet method. Multiscale Model. Simul. **5**(3), 980–995 (2006)
3. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial: Second Edition. SIAM, Philadelphia, PA (2000)
4. Chaniotis, A.K., Poulikakos, D., Koumoutsakos, P.: Remeshed smoothed particle hydrodynamics for the simulation of viscous and heat conducting flows. J. Comput. Phys. **182**(1), 67–90 (2002)
5. Cottet, G.H., Koumoutsakos, P.: Vortex Methods, Theory and Practice. Cambridge University Press (2000)
6. Cottet, G.H., Maitre, E.: A level set method for fluid-structure interactions with immersed surfaces. Math. Models Methods Appl. Sci. **16**(3), 415–438 (2006)
7. Foster, N., Metaxas, D.: Controlling fluid animation. In: Proceedings CGI ’97, pp. 178–188 (1997)
8. Georgii, J., Westermann, R.: Mass-spring systems on the GPU. Simul. Model. Pract. Theory **13**(8), 693–702 (2005)
9. Hagen, T.R., Lie, K.A., Natvig, J.R.: Solving the Euler equations on graphics processing units. Comput. Sci. (ICCS 2006) **3994**, 220–227 (2006)

10. Harada, T., Koshizuka, S., Kawaguchi, Y.: Smoothed particle hydrodynamics on GPUs. In: Proc. of Computer Graphics International, pp. 63–70 (2007)
11. Harris, M.J.: Fast fluid dynamics simulation on the GPU. In: GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, pp. 637–665. Addison-Wesley (2004)
12. Kolb, A., Cuntz, N.: Dynamic particle coupling for GPU-based fluid simulation. In: Proc. ASIM, pp. 722–727 (2005)
13. Kolb, A., Latta, L., Rezk-Salama, C.: Hardware-based simulation and collision detection for large particle systems. In: Proc. Graphics Hardware, pp. 123–131. ACM/Eurographics, Grenoble, France (2004)
14. Koumoutsakos, P.: Inviscid axisymmetrization of an elliptical vortex. J. Comput. Phys. **138**(2), 821–857 (1997)
15. Koumoutsakos, P.: Multiscale flow simulations using particles. Annu. Rev. Fluid Mech. **37**, 457–487 (2005)
16. Krüger, J., Schiwietz, T., Kipfer, P., Westermann, R.: Numerical simulations on PC graphics hardware. In: ParSim 2004 (Special Session of EuroPVM/MPI 2004) (2004)
17. Hegeman, K., Carr, N.A., Miller, G.S.: Particle-based fluid simulation on the GPU. In: Computational Science – ICCS 2006, vol. 3994, pp. 228–235. Springer, Berlin/Heidelberg (2006)
18. Minion, M.L., Brown, D.L.: Performance of under-resolved two-dimensional incompressible flow simulations, ii. J. Comput. Phys. **138**, 734–765 (1997)
19. Monaghan, J.J.: Smoothed particle hydrodynamics. Rep. Prog. Phys. **68**(8), 1703–1759 (2005)
20. Müller, M., Charypar, D., Gross, M.: Particle-based fluid simulation for interactive applications. In: SCA ’03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 154–159. Eurographics Association, San Diego, CA (2003)
21. Pfister, H., Gross, M.: Point-based computer graphics. IEEE Comput. Graph. Appl. **24**(4), 22–23 (2004)
22. Scheidegger, C.E., Comba, J.L.D., da Cunha, R.D.: Practical CFD simulations on programmable graphics hardware using smac. Comput. Graph. Forum **24**(4), 715–728 (2005)
23. Selle, A., Rasmussen, N., Fedkiw, R.: A vortex particle method for smoke, water and explosions. ACM Trans. Graph. **24**(3), 910–914 (2005). http://doi.acm.org/10.1145/1073204.1073282
24. Stam, J.: A simple fluid solver based on the FFT. J. Graph. Tools **6**(2), 43–52 (2001)
25. Treuille, A., Lewis, A., Popovic, Z.: Model reduction for real-time fluids. ACM Trans. Graph. **25**(3), 826–834 (2006)
26. Wu, E.H., Zhu, H.B., Liu, X.H., Liu, Y.Q.: Simulation and interaction of fluid dynamics. Visual Comput. **23**(5), 299–308 (2007)

DIEGO ROSSINELLI was born in 1982. He received his master’s degree in Computer Science in 2006 from ETHZ, Switzerland. Since 2006, he is a Ph.D. student at the Chair of Computational Science, ETHZ, Switzerland. His field of interests includes real-time simulations of fluids and high performance computing.

PETROS KOUMOUTSAKOS received his Ph.D. in Aeronautics and Applied Mathematics at the California Institute of Technology in 1992. He holds the Chair of Computational Science (2000–) in the Department of Computer Science and he is accredited with the Department of Mechanical and Process Engineering at ETHZ. His professional interests are in the development of computational methods for the study of diverse problems in Engineering and Life Sciences. He leads a group of Ph.D. students, post-doctoral fellows and visiting scientists working in multiscale modeling and simulation, high performance computing, bioinspired optimization and design. Example applications range from the reverse engineering of swimming devices, to the design of nanosyringes and the modeling of tumour induced angiogenesis.