Leapfrog Flow Maps for Real-Time Fluid Simulation

YUCHEN SUN, Georgia Institute of Technology, USA
JUNLIN LI, Georgia Institute of Technology, USA
RUICHENG WANG, Georgia Institute of Technology, USA
SINAN WANG, Georgia Institute of Technology, USA
ZHIQI LI, Georgia Institute of Technology, USA
BART G. VAN BLOEMEN WAANDERS, Sandia National Laboratories, USA
BO ZHU, Georgia Institute of Technology, USA

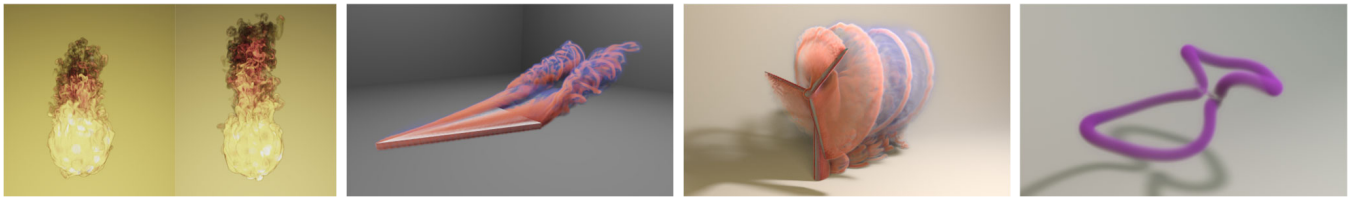


Fig. 1. Burning fire ball (left), delta wingtip vortices (middle left), helical trail behind a wind turbine (middle right), two connecting vortex rings (right).

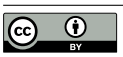
We propose Leapfrog Flow Maps (LFM) to simulate incompressible fluids with rich vortical flows in real time. Our key idea is to use a hybrid velocity-impulse scheme enhanced with leapfrog method to reduce the computational workload of impulse-based flow map methods, while possessing strong ability to preserve vortical structures and fluid details. In order to accelerate the impulse-to-velocity projection, we develop a fast matrix-free Algebraic Multigrid Preconditioned Conjugate Gradient (AMGPCG) solver with customized GPU optimization, which makes projection comparable with impulse evolution in terms of time cost. We demonstrate the performance of our method and its efficacy in a wide range of examples and experiments, such as real-time simulated burning fire ball and delta wingtip vortices.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: Real-Time Fluid Simulation Leapfrog Integration, Flow Map Methods, Impulse Gauge Method, Multigrid, GPU

ACM Reference Format:
Yuchen Sun, Junlin Li, Ruicheng Wang, Sinan Wang, Zhiqi Li, Bart G. van Bloemen Waanders, and Bo Zhu. 2025. Leapfrog Flow Maps for Real-Time Fluid Simulation. *ACM Trans. Graph.* 44, 4 (August 2025), 12 pages. https://doi.org/10.1145/3731180

Authors' Contact Information: Yuchen Sun, yuchen.sun.eecs@gmail.com, Georgia Institute of Technology, USA; Junlin Li, jli3518@gatech.edu, Georgia Institute of Technology, USA; Ruicheng Wang, wrc0326@outlook.com, Georgia Institute of Technology, USA; Sinan Wang, swang3081@gatech.edu, Georgia Institute of Technology, USA; Zhiqi Li, zli3167@gatech.edu, Georgia Institute of Technology, USA; Bart G. van Bloemen Waanders, bartv@sandia.gov, Sandia National Laboratories, USA; Bo Zhu, bo.zhu@gatech.edu, Georgia Institute of Technology, USA.



ACM 1557-7368/2025/8-ART
https://doi.org/10.1145/3731180

1 Introduction

Producing and evolving vortical structures in a real-time simulation environment has been a challenging task. Existing real-time systems predominantly rely on a combination of stable fluid solvers [Stam 1999a] on grids, enhanced by vorticity confinement techniques [Selle et al. 2005] or noise functions, to synthesize small-scale details into a flow field (e.g., see the fluid simulation solvers in real-time physics engines such as EmberGen [JangaFX 2024] and Unreal [Games 2024]). These methods have achieved significant success in generating visually appealing fluid motions and enabling interactions with objects in real time.

However, despite the impressive visual effects of these small-scale turbulent details, producing physically plausible or accurate vortical evolution in real time remains challenging. This limitation constrains the applications of real-time fluid solvers to the realm of visual effects. In applications requiring a certain degree of physical fidelity, these confinement-based or noise-based approaches reveal their shortcomings. For example, existing fast fluid solvers often struggle to successfully pass benchmark vorticity evolution tests, such as producing a Kármán vortex street or simulating the leapfrog evolution of vortex rings.

This paper takes an initial step toward developing a real-time fluid solver capable of simulating physically accurate vortical structure evolution. The solver is built upon the impulse-based flow-map framework, which has garnered increasing attention in recent years due to its inherent ability to preserve spatiotemporal vortical structures in various fluid simulation tasks. A key mechanism that enables flow-map methods to preserve vortical structures is their bidirectional marching scheme, which allows a particle to move forward and backward along its flow-map trajectory in a temporally symmetric manner, ensuring the conservation of important geometric and physical quantities.

However, establishing such a bidirectional map is computationally expensive and demands significant memory resources. In Neural

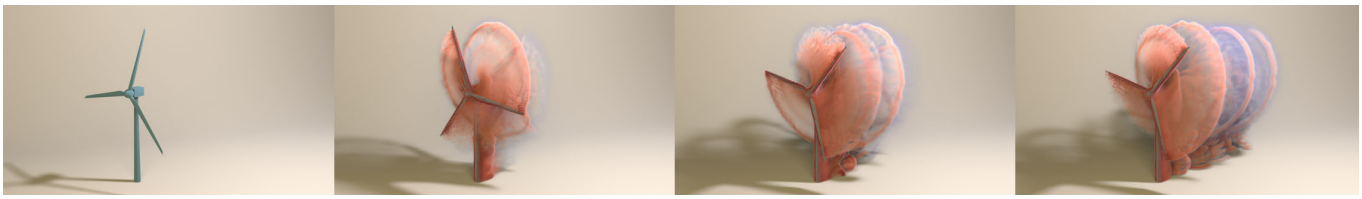


Fig. 2. Wind Turbine. A wind turbine rotates in an inlet. The blades of the wind turbine generate helical trails downstream.

Flow Maps (NFM) [Deng et al. 2023], creating the bidirectional map requires maintaining a velocity buffer with snapshots of the velocity field from previous time frames. This accommodates long-distance backtracing from the current grid nodes to their initial mapping starting points. In particular, calculating the mapping for the k-th time step along the map trajectory requires storing $O(k)$ velocity buffers and backtrace with $O(k)$ time steps to get to the initial point. Second, to achieve symmetric time integration for each time step along the flow map, an additional projection step is required to generate a divergence-free velocity at the midpoint between t_n and t_{n+1}. This doubles the projection cost compared to a standard grid-based advection-projection solver. Consequently, the time performance of flow-map methods, as reported in the current literature, falls significantly short of the standards required for a real-time fluid solver.

We addressed the two primary challenges hindering the advancement of flow-map methods toward real-time simulation by introducing a novel backtrace time integrator. Inspired by the Leapfrog time integrator from the ODE literature, which is widely used to evolve symplectic systems in a geometrically symmetric manner (i.e., forward and backward motion along the time axis remains on the same trajectory), we developed Leapfrog Flow Maps (LFM). This approach significantly reduces the number of backtrace steps and simplifies the projection steps.

Our key observation is that the accuracy of the mapped impulse at the end of a reinitialization cycle is critical, as it solely determines the future evolution of the flow. However, backtracing for impulse may not be necessary during intermediate steps, providing an opportunity to optimize the previously complex time integrator. Based on this observation, we propose a hybrid velocity-impulse scheme. This scheme employs velocity advection, enhanced by the Leapfrog time integrator, for intermediate steps, combined with a long-range, accurate bidirectional marching step at the end of each reinitialization cycle. Furthermore, the interleaved advection in the Leapfrog method eliminates the need for extra projection steps while maintaining second-order accuracy.

Our experiments demonstrate that the LFM scheme achieves comparable simulation accuracy to the traditional flow-map methods (e.g., NFM [Deng et al. 2023]) while significantly reducing computational workload, making it more suitable for real-time applications.

We summarize our main contributions as:

- Leapfrog Flow Maps (LFM), a hybrid velocity-impulse time integrator to reduce the projection times by half and the flow-map marching steps by up to 76% in NFM.
- A fast matrix-free Algebraic Multigrid Preconditioned Conjugate Gradient (AMGPCG) solver on GPU to solve large-scale Poisson systems for real-time fluid simulation.
- A real-time fluid simulator to produce accurate and interactive vortical evolution at the resolution of $256 \times 128 \times 128$.

2 Related Work

Gauge Methods. Impulse can be defined by rewriting the Navier-Stokes equations into the Hamiltonian formulation through a gauge transformation [Oseledets 1989]. Researchers in the field of Computational Fluid Dynamics initially employed impulse to preserve the Hamiltonian structures of incompressible fluid flows [Buttke 1992]. Subsequent investigations have explored various aspects of impulse, including turbulence [Buttke and Chorin 1993], boundary force [Cortez 1996], solid boundary treatment [Summers 2000] and numerical stability [Weinan and Liu 2003]. Saye [2016, 2017] developed a discontinuous Galerkin method for simulating multiphase flow, based on a simplified velocity-impulse density formulation [Weinan and Liu 1997]. More recently, the concept of impulse has garnered increasing attention within the computer graphics community [Feng et al. 2023; Yang et al. 2021]. Nabizadeh et al. [2022] combined impulse with bidirectional flow maps [Qu et al. 2019] to better capture vortical structures. Deng et al. [2023] proposed a backward marching scheme with a neural buffer to improve the accuracy of flow map tracking and impulse transport. Beyond Eulerian approaches, hybrid Eulerian-Lagrangian methods for impulse have been developed [Nabizadeh et al. 2024; Zhou et al. 2024]. In addition to smoke simulation, impulse-based methods have been applied to free-surface flows [Li et al. 2024a; Sancho et al. 2024], two-phase flows [Sun et al. 2024], solid-fluid interaction [Chen et al. 2024] and particle-laden fluid [Li et al. 2024b].

Multigrid. Projection is the bottleneck of fluid simulation. Some researchers accelerate projection via multigrid methods. McAdams et al. [2010] proposed a Geometric Multigrid method supporting mixed Neumann and Dirichlet boundary conditions. Their method can be massively parallelized, but it cannot handle Poisson equation with variable coefficients (VC) and only has first-order accuracy for boundary condition. Chentanez and Müller [2011] devised a GPU-friendly multigrid method for liquid simulation on a tall cell grid. Instead of coarsening, they compute the coefficients for each level separately based on level set and solid fraction, which fails to satisfy the Galerkin Principle. Conversely, Algebraic Multigrid (AMG) [Ruge and Stüben 1987] has second-order accuracy and a Galerkin coarsening strategy. Recently, researchers have explored using AMG to efficiently solve Poisson equations with non-negative

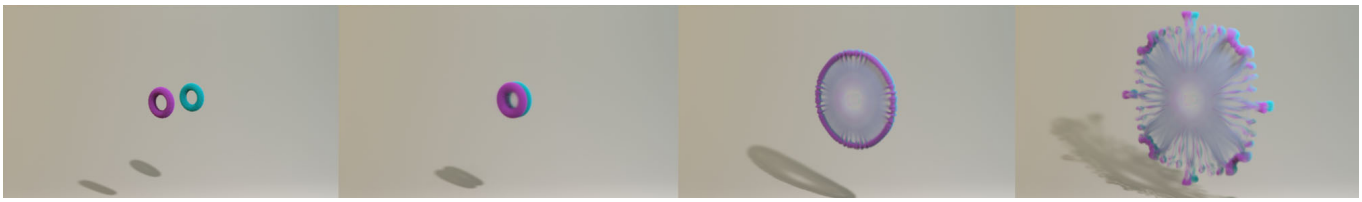


Fig. 3. Head-on Vortex Rings Collision. A pair of vortex rings with opposite vorticity approach each other, then collide.

pressure constraints in fluid simulation [Takahashi and Batty 2023, 2025]. A variant of Algebraic Multigrid, Unsmoothed Aggregation Algebraic Multigrid (UAAMG), can be implemented in a matrix-free manner on a grid. Although UAAMG converges more slowly than the original AMG, adding a scale coefficient in the prolongation step can improve convergence when UAAMG is used as a preconditioner for the Conjugate Gradient solver [Stüben 2001]. UAAMG was first applied to fluid simulation by Bolz et al. [2003] and was adopted in the smoke simulator of Houdini [Zarifi 2020]. Shao et al. [2022] extended it to viscous liquid simulation and provided a fast matrix-free CPU implementation accelerated by Advanced Vector Extensions (AVX) instructions. Naumov et al. [2015] and Bernaschi et al. [2020] provided efficient libraries of GPU AMG solvers. However, these libraries are based on sparse matrix representation which inevitably incurs huge performance cost [Shao et al. 2022]. By contrast, our solver is matrix-free and highly optimized via techniques such as tiled data structure, data trimming and aggregated CUDA kernels.

Table 1. A comparison between previous Multigrid solvers and ours.

Method	GPU	VC	matrix-free	Galerkin
[McAdams et al. 2010]	✗	✗	✓	✗
[Chentanez and Müller 2011]	✓	✓	✓	✗
[Naumov et al. 2015]	✓	✓	✗	✓
[Bernaschi et al. 2020]	✓	✓	✗	✓
[Shao et al. 2022]	✗	✓	✓	✓
Ours	✓	✓	✓	✓

High-Performance Eulerian Fluid Simulation. In recent years, a wide range of approaches have been explored to enhance the performance of Eulerian fluid simulation, including adaptive solvers, sparse solvers, spectral methods, data-driven methods and Lattice Boltzman methods (LBM). The use of spatial adaptivity to reduce computational workload has been extensively studied since the pioneering work of Berger and Oliger [1984]. Two widely adopted adaptive grid data structures are octrees [Ando and Batty 2020; Goldade et al. 2019; Losasso et al. 2004] and sparse paged grids [Aanjaneya et al. 2017; Liu et al. 2016; Setaluri et al. 2014]. Instead of a single adaptive grid, Lentine et al. [2010] proposed utilizing overlapping grids with different resolutions, as the irregular data structures in adaptive grids hinder GPU acceleration. In liquid simulation, fluids typically occupy only a small region of the computational domain, which can be efficiently simulated with sparse volume structures [Museth 2013; Wu et al. 2018]. For smoke simulation with specific boundary condition, the spectral modes and eigenfunctions of the Laplacian operator can be used to accelerate the simulation performance [Cui et al. 2018; Stam 1999b]. Motivated by spectral methods, Rabbani et al. [2022] proposed an efficient Poisson-filter solver capable of approximating multiple Jacobi iterations. With the rise of deep learning, some researchers have explored the use of neural networks for pressure projection to achieve performance improvement [Tompson et al. 2017; Yang et al. 2016]. While the previously mentioned methods focus on improving pressure projection, Lattice Boltzmann methods have the advantage of not requiring pressure projection in their formulation. Researchers have developed efficient Lattice Boltzman methods for fluid-solid coupling [Li et al. 2020; Lyu et al. 2021], two-phase flows [Li et al. 2022] and virtual wind tunnels [Lyu et al. 2023].

Notation	Type	Meaning
ρ	scalar	density
$\boldsymbol{u}$	vector	velocity
p	scalar	pressure
μ	scalar	dynamic viscosity
$\boldsymbol{f}$	vector	external force
$\boldsymbol{m}$	vector	impulse
$\boldsymbol{\Phi}$	vector	forward flow map
$\boldsymbol{\Psi}$	vector	backward flow map
$\mathcal{F}$	matrix	Jacobian of forward flow map
$\mathcal{T}$	matrix	Jacobian of backward flow map
ξ	scalar	gauge variable
n	scalar	number of steps in a reinitialization cycle

Table 2. Summary of notations in this paper.

3 Physical Model

We use regular symbols for scalars, bold symbols for vectors, and calligraphic symbols for matrices. The temporal parameter of a field is placed in the subscript, while the spatial parameter is placed in parentheses. The notations used throughout this paper are summarized in Table 2.

Consider the incompressible Navier-Stokes equations:

$$\begin{cases} \rho \dfrac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} = -\nabla p + \mu \nabla^2 \boldsymbol{u} + \boldsymbol{f}, \\ \nabla \cdot \boldsymbol{u} = 0. \end{cases} \tag{1}$$

where $\boldsymbol{f}$ denotes external forces such as gravity and buoyancy. Based on the velocity field $\boldsymbol{u}$, the impulse field $\boldsymbol{m}$ is defined through its

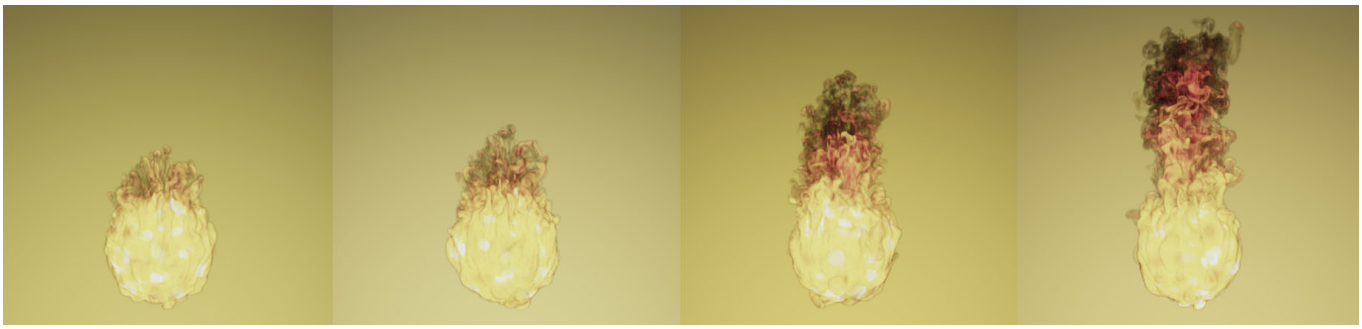


Fig. 4. Fire Ball. A burning fireball surrounded by swirling vortical flames, where the intense combustion generates complex vortex structures that propagate outward with the flow of hot gases.

initial condition and material derivative:

$$\begin{cases} \boldsymbol{m}_0(\boldsymbol{x}) = \boldsymbol{u}_0(\boldsymbol{x}), \\ \dfrac{D\boldsymbol{m}}{Dt} = -(\nabla \boldsymbol{u})^T \boldsymbol{m}. \end{cases} \tag{2}$$

We utilize flow maps to describe the motion of fluids. Consider a fluid particle moving according to a flow field $\boldsymbol{u}$. Over the time interval from a to b, the forward flow map $\boldsymbol{\Phi}_{a,b}$ maps the initial position $\boldsymbol{X}$ of a fluid particle to its final position $\boldsymbol{x}$. The backward flow map $\boldsymbol{\Psi}_{b,a}$ is defined as the inverse mapping of $\boldsymbol{\Phi}_{a,b}$.

$$\begin{cases} \boldsymbol{\Phi}_{a,b}(\boldsymbol{X}) = \boldsymbol{x}, \\ \boldsymbol{\Psi}_{b,a}(\boldsymbol{x}) = \boldsymbol{X}. \end{cases} \tag{3}$$

The Jacobians of $\boldsymbol{\Phi}_{a,b}$ and $\boldsymbol{\Psi}_{b,a}$ are represented by $\boldsymbol{\mathcal{F}}_{a,b}$ and $\boldsymbol{\mathcal{T}}_{b,a}$ respectively. From a flow-map perspective, impulse can be reconstructed based on its initial condition and the backward flow map through the pullback operator [Nabizadeh et al. 2022]:

$$\boldsymbol{m}_t(\boldsymbol{x}) = \boldsymbol{\mathcal{T}}^T_{t,0}(\boldsymbol{x}) \boldsymbol{u}_0(\boldsymbol{\Psi}_{t,0}(\boldsymbol{x})). \tag{4}$$

The difference between the impulse and velocity fields consists of a gauge term and the path integral of viscous and external forces along the trajectory of a fluid particle:

$$\begin{aligned} \boldsymbol{u}_t(\boldsymbol{x}) = \boldsymbol{m}_t(\boldsymbol{x}) - \frac{1}{\rho} \nabla \xi_{0,t}(\boldsymbol{x}) + \\ \frac{1}{\rho} \boldsymbol{\mathcal{T}}^T_{t,0}(\boldsymbol{x}) \int_0^t \boldsymbol{\mathcal{F}}^T_{0,\tau}(\boldsymbol{X}) \left(\mu \nabla^2 \boldsymbol{u}_\tau + \boldsymbol{f}_\tau \right) (\boldsymbol{\Psi}_{t,\tau}(\boldsymbol{x})) \mathrm{d}\tau. \end{aligned} \tag{5}$$

Here $\boldsymbol{X}$ is the initial position of a fluid particle that is located at $\boldsymbol{x}$ at time t, and $\boldsymbol{\Psi}_{t,\tau}(\boldsymbol{x})$ specifies the position of the fluid particle at time τ. We have $\xi_{0,t}$ representing the gauge variable as:

$$\xi_{0,t}(\boldsymbol{x}) = \int_0^t \left(p_\tau - \frac{1}{2} \rho |\boldsymbol{u}_\tau|^2 \right) (\boldsymbol{\Psi}_{t,\tau}(\boldsymbol{x})) \mathrm{d}\tau \tag{6}$$

The velocity field can be obtained from the impulse field by first subtracting the path integral term from the impulse, followed by solving a Poisson equation for the gauge variable.

4 Leapfrog Flow Maps

NFM [Deng et al. 2023] has demonstrated high simulation fidelity and excellent vortex preservation. However, its complex flow-map time integration scheme is significantly slower compared to traditional velocity-based simulators. Specifically, at every time step, NFM has an extra projection for the midpoint velocity and involves multiple marching steps of backward flow maps to pull back the impulse. Drawing inspiration from both NFM and the Leapfrog Method, we propose Leapfrog Flow Maps (LFM), a much more efficient hybrid velocity-impulse approach that achieves comparable results.

For simplicity in the following discussion, we abbreviate the end time t_i of step i as i in subscripts.

4.1 Reinitialization Cycle

Like other impulse-based flow map methods [Deng et al. 2023; Nabizadeh et al. 2022], LFM requires periodic reinitialization of the impulse as velocity and the flow map as the identity map every certain number of time steps to prevent numerical instability caused by flow map distortion. Consider a reinitialization cycle with n time steps. Across this time interval, the initial velocity $\boldsymbol{u}_0$ is transformed to impulse $\boldsymbol{m}_n$ via the long-range backward flow map $\boldsymbol{\Psi}_{n,0}$ (Equation (4)). Then, $\boldsymbol{m}_n$ is projected onto $\boldsymbol{u}_n$ by solving a Poisson equation, and $\boldsymbol{u}_n$ becomes the initial velocity for the subsequent reinitialization cycle. Instead of advection, which involves frequent interpolation and significant numerical dissipation, we adopt marching to obtain $\boldsymbol{\Psi}_{n,0}$. In each step, we directly store the velocity fields of different time steps in the global memory of GPU rather than using a neural buffer [Deng et al. 2023] for performance consideration. The global memory buffer is cleared when reinitialization happens. After the velocity fields are available, we can directly backward march a virtual fluid particle located at a grid point to compute its trajectory. We employ the RK4 marching customized for flow maps and their Jacobians (Algorithm 2 of NFM [Deng et al. 2023]) to evolve $\boldsymbol{\Psi}$ and $\boldsymbol{\mathcal{T}}$. Quadratic B splines is used for velocity interpolation. While the marching scheme provides an accurate backward flow map, interpolation during the pullback process may introduce numerical dissipation. To alleviate this, we simultaneously march the forward flow map and its Jacobian at each step and use them for back-and-forth error compensation.

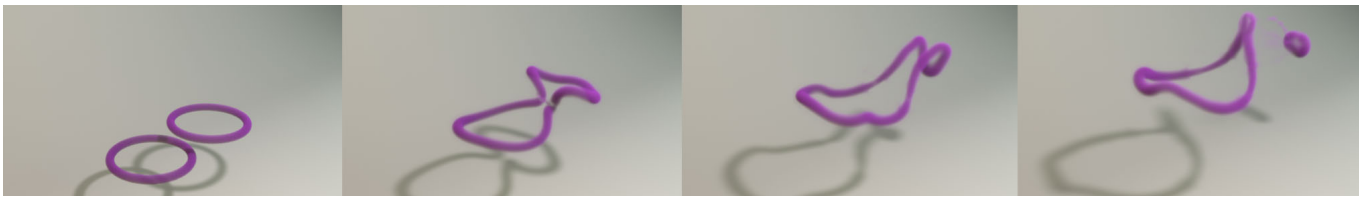


Fig. 5. Vortex Rings Connection. A pair of vortex rings deform, connect. A small vortex ring then separate off.

Algorithm 1 LFM Reinitialization Cycle

Input: $\boldsymbol{u}_0$
Output: $\boldsymbol{u}_n$

$\boldsymbol{u}_0^\dagger \leftarrow \boldsymbol{u}_0 + \frac{\Delta t}{2\rho}(\mu\nabla^2\boldsymbol{u}_0 + \boldsymbol{f}_0)$; ▷ blue for viscosity and ext force
$\tilde{\boldsymbol{u}}_{1/2} \leftarrow$ **RK2-Advect**$(\boldsymbol{u}_0^\dagger, \boldsymbol{u}_0, \Delta t/2)$;
$\boldsymbol{u}_{1/2} \leftarrow$ **Project**$(\tilde{\boldsymbol{u}}_{1/2})$;Store $\boldsymbol{u}_{1/2}$ in buffer;
$\Phi_{0,1}, \mathcal{F}_{0,1} \leftarrow$ **RK4-March**$(\Phi_{0,0}, \mathcal{F}_{0,0}, \boldsymbol{u}_{1/2}, \Delta t)$;
$\boldsymbol{u}_0 \leftarrow \boldsymbol{u}_0 + \frac{\Delta t}{\rho}\mathcal{F}^T_{0,1/2}(\mu\nabla^2\boldsymbol{u}_{1/2} + \boldsymbol{f}_{1/2})(\Phi_{0,1/2})$;
$\boldsymbol{u}_{1/2}^\dagger \leftarrow \boldsymbol{u}_{1/2} + \frac{\Delta t}{\rho}(\mu\nabla^2\boldsymbol{u}_{1/2} + \boldsymbol{f}_{1/2})$;
$\tilde{\boldsymbol{u}}_{3/2} \leftarrow$ **RK2-Advect**$(\boldsymbol{u}_{1/2}^\dagger, \boldsymbol{u}_{1/2}, \Delta t)$;
$\boldsymbol{u}_{3/2} \leftarrow$ **Project**$(\tilde{\boldsymbol{u}}_{3/2})$; Store $\boldsymbol{u}_{3/2}$ in buffer;
$\Phi_{0,2}, \mathcal{F}_{0,2} \leftarrow$ **RK4-March**$(\Phi_{0,1}, \mathcal{F}_{0,1}, \boldsymbol{u}_{3/2}, \Delta t)$;
$\boldsymbol{u}_0 \leftarrow \boldsymbol{u}_0 + \frac{\Delta t}{\rho}\mathcal{F}^T_{0,3/2}(\mu\nabla^2\boldsymbol{u}_{3/2} + \boldsymbol{f}_{3/2})(\Phi_{0,3/2})$;
for $i = 2$ to $n-1$ **do**
  $\boldsymbol{u}_{i-3/2}^\dagger \leftarrow \boldsymbol{u}_{i-3/2} + \frac{2\Delta t}{\rho}(\mu\nabla^2\boldsymbol{u}_{i-1/2} + \boldsymbol{f}_{i-1/2})$;
  $\tilde{\boldsymbol{u}}_{i+1/2} \leftarrow$ **RK2-Advect**$(\boldsymbol{u}_{i-3/2}^\dagger, \boldsymbol{u}_{i-1/2}, 2\Delta t)$;
  $\boldsymbol{u}_{i+1/2} \leftarrow$ **Project**$(\tilde{\boldsymbol{u}}_{i+1/2})$;Store $\boldsymbol{u}_{i+1/2}$ in buffer;
  $\Phi_{0,i+1}, \mathcal{F}_{0,i+1} \leftarrow$ **RK4-March**$(\Phi_{0,i}, \mathcal{F}_{0,i}, \boldsymbol{u}_{i+1/2}, \Delta t)$;
  $\boldsymbol{u}_0 \leftarrow \boldsymbol{u}_0 + \frac{\Delta t}{\rho}\mathcal{F}^T_{0,i+1/2}(\mu\nabla^2\boldsymbol{u}_{i+1/2} + \boldsymbol{f}_{i+1/2})(\Phi_{0,i+1/2})$;
end for
$\Psi_{n,n} \leftarrow$ **id**, $\Phi_{0,0} \leftarrow$ **id**, $\mathcal{T}_{n,n} \leftarrow \boldsymbol{I}$, $\mathcal{F}_{0,0} \leftarrow \boldsymbol{I}$;
for $i = n$ to 1 **do**
  $\Psi_{n,i-1}, \mathcal{T}_{n,i-1} \leftarrow$ **RK4-March**$(\Psi_{n,i}, \mathcal{T}_{n,i}, \boldsymbol{u}_{i-1/2}, -\Delta t)$;
end for
$\boldsymbol{m}_n \leftarrow \mathcal{T}^T_{n,0}\boldsymbol{u}_0(\Psi_{n,0})$;
$\hat{\boldsymbol{u}}_0 \leftarrow \mathcal{F}^T_{0,n}\boldsymbol{m}_n(\Phi_{0,n})$;
$\boldsymbol{e} \leftarrow (\hat{\boldsymbol{u}}_0 - \boldsymbol{u}_0)/2$; ▷ roundtrip error
$\boldsymbol{m}_n \leftarrow \boldsymbol{m}_n - \mathcal{T}^T_{n,0}\boldsymbol{e}(\Psi_{n,0})$; ▷ error compensation
$\boldsymbol{m}_n \leftarrow$ **Clamp**$(\boldsymbol{m}_n)$ ▷ optional
$\boldsymbol{u}_n \leftarrow$ **Project**$(\boldsymbol{m}_n)$;

4.2 Leapfrog Method for Midpoint Velocity

The Leapfrog method is a second-order symplectic integrator with only modest additional computational cost compared to the first-order Euler method, while providing better long-term stability. Another type of symplectic integrator, Verlet method, has been widely used in Position Based Dynamics [Jakobsen 2001]. Using the midpoint velocity at each time step ensures second-order accuracy for each marching step. Additionally, it maintains the symmetry between the backward and forward flow map evolutions, further enhancing the effectiveness of back-and-forth error compensation.

While the eventual velocity $\boldsymbol{u}_n$ is obtained from a long-ranged mapped impulse, the midpoint velocities are computed via velocity advection, which is significantly more efficient.

We employ the leapfrog method combined with RK2 advection to compute the midpoint velocities, ensuring second-order accuracy, and require a consistent step size Δt for all time steps within each reinitialization cycle. In the leapfrog method, velocities are advected in an interleaved manner, where the projection time of the previous step serves as the midpoint for the next advection. This approach enables second-order accuracy in velocity advection without requiring additional computations to explicitly evaluate the midpoint velocity for the next step. A special case arises during the first two steps of the reinitialization cycle, where the leapfrog scheme cannot be applied because the required midpoint velocities from previous steps are unavailable.

Once the midpoint velocities are computed, we perform a backward marching to obtain $\Psi_{n,0}$ and subsequently $\boldsymbol{u}_n$. The velocity $\boldsymbol{u}_n$ solely determines the fluid's future evolution after reinitialization at time step n, while the midpoint velocities serve only as auxiliary variables for computing $\boldsymbol{u}_n$. In experiments, we find that the midpoint method, impulse, long-range flow maps, and error compensation are all essential for accurately determining $\boldsymbol{u}_n$. The midpoint velocities contribute to the calculation of $\boldsymbol{u}_n$ through the marching of bidirectional flow maps. Additionally, we observe that leapfrog velocity advection for computing the midpoint velocities yields favorable results for $\boldsymbol{u}_n$. Leveraging impulse is unnecessary for computing the midpoint velocities.

4.3 Comparison with NFM

We compare the simulation part of NFM with LFM, excluding the neural buffer. The pseudocode for an LFM reinitialization cycle and an NFM reinitialization cycle are presented in Algorithm 1 and Algorithm 2, respectively. LFM has a significantly lower computational workload compared to NFM. While NFM achieves high simulation accuracy, it requires two projections and multiple marching steps to compute flow maps at each time step. In contrast, LFM only needs one projection and one advection per dimension in each time step. Despite this, our experiments show that LFM achieves simulation accuracy comparable to NFM.

NFM stores flow maps on each face of a staggered grid. For a reinitialization cycle with n time steps, NFM requires $\frac{1}{2}n^2 + \frac{5}{2}n$ marching steps per dimension and $2n$ projections. In comparison, LFM only requires n advections, $2n$ marching steps per dimension, and $n+1$ projections for the same reinitialization cycle. Moreover, marching

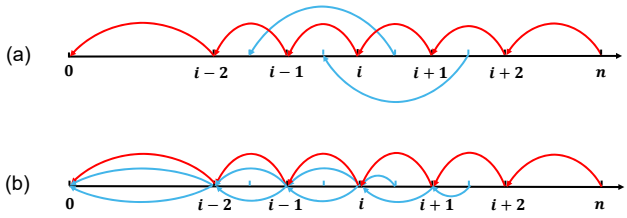


Fig. 6. Illustration for advections and marchings in one reinitialization cycle. The blue arrows show the advection / marching trajectory for midpoint velocities. The red arrows show marching trajectory for $\boldsymbol{u}_n$. (a) LFM (b) NFM.

Algorithm 2 NFM Reinitialization Cycle

Input: $\boldsymbol{u}_0$
Output: $\boldsymbol{u}_n$

$\boldsymbol{\Phi}_{0,0} \leftarrow \mathbf{id}$, $\boldsymbol{\mathcal{F}}_{0,0} \leftarrow \boldsymbol{I}$;
for $i = 1$ to n **do**
  $\boldsymbol{\Psi}_{i-1/2,i-1}, \boldsymbol{\mathcal{T}}_{i-1/2,i-1} \leftarrow$ **RK4-March**$(\mathbf{id}, \boldsymbol{I}, \boldsymbol{u}_{i-1}, -\Delta t/2)$;
  $\boldsymbol{u}_{i-1/2} \leftarrow$ **Project**$(\boldsymbol{\mathcal{T}}^T_{i-1/2,i-1}\boldsymbol{u}_{i-1}(\boldsymbol{\Psi}_{i-1/2,i-1}))$;
  Store $\boldsymbol{u}_{i-1/2}$ in buffer; ▷ midpoint method
  $\boldsymbol{\Phi}_{0,i}, \boldsymbol{\mathcal{F}}_{0,i} \leftarrow$ **RK4-March**$(\boldsymbol{\Phi}_{0,i-1}, \boldsymbol{\mathcal{F}}_{0,i-1}, \boldsymbol{u}_{i-1/2}, \Delta t)$;
  $\boldsymbol{\Psi}_{i,i} \leftarrow \mathbf{id}$, $\boldsymbol{\mathcal{T}}_{i,i} \leftarrow \boldsymbol{I}$;
  for $j = i$ to 1 **do**
    $\boldsymbol{\Psi}_{i,j-1}, \boldsymbol{\mathcal{T}}_{i,j-1} \leftarrow$ **RK4-March**$(\boldsymbol{\Psi}_{i,j}, \boldsymbol{\mathcal{T}}_{i,j}, \boldsymbol{u}_{j-1/2}, -\Delta t)$
  end for
  $\boldsymbol{m}_i \leftarrow \boldsymbol{\mathcal{T}}^T_{i,0}\boldsymbol{u}_0(\boldsymbol{\Psi}_{i,0})$;
  $\hat{\boldsymbol{u}}_0 \leftarrow \boldsymbol{\mathcal{F}}^T_{0,i}\boldsymbol{m}_i(\boldsymbol{\Phi}_{0,i})$;
  $\boldsymbol{e} \leftarrow (\hat{\boldsymbol{u}}_0 - \boldsymbol{u}_0)/2$; ▷ roundtrip error
  $\boldsymbol{m}_i \leftarrow \boldsymbol{m}_i - \boldsymbol{\mathcal{T}}^T_{i,0}\boldsymbol{e}(\boldsymbol{\Psi}_{i,0})$ ▷ error compensation
  $\boldsymbol{m}_i \leftarrow$ **Clamp**$(\boldsymbol{m}_i)$ ▷ optional
  $\boldsymbol{u}_i \leftarrow$ **Project**$(\boldsymbol{m}_i)$;
end for

is computationally more expensive than advection because it involves additional calculations, such as evaluating velocity gradients and updating flow map Jacobians.

In a reinitialization cycle, LFM reduces the number of projections by nearly half compared to NFM. While the number of marching steps in NFM grows quadratically with n, the advection and marching steps in LFM scale linearly with n. As illustrated in Fig. 6, LFM requires significantly fewer advection and marching steps than NFM within a reinitialization cycle. For the longest reinitialization length of $n = 20$ used by NFM, NFM requires 250 marching steps per dimension, whereas LFM only needs 20 advection steps and 40 marching steps per dimension—a reduction over 76%.

4.4 Viscosity and External Force

Another limitation of NFM is its inability to simulate real fluids influenced by viscosity and external forces. When viscosity and external forces affect fluid motion, the impulse cannot be directly projected onto velocity due to the appearance of an additional term in the difference between impulse and velocity (as shown in Equation (5)). To address this, the additional term must be evaluated and added

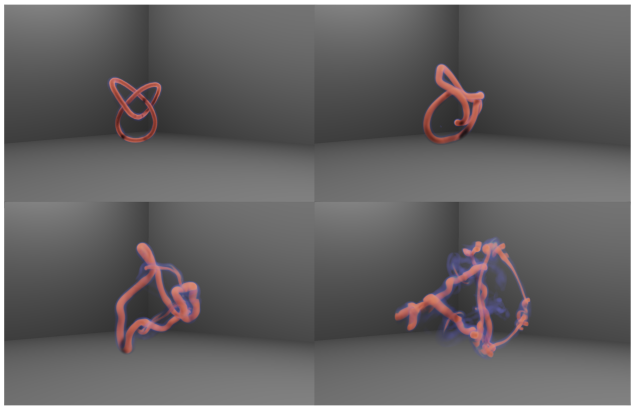


Fig. 7. Trefoil. A trefoil knot breaks into a large vortex and a small vortex.

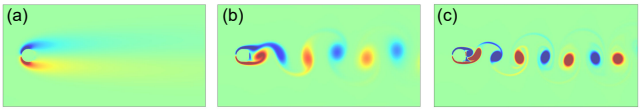


Fig. 8. Kármán Vortex Street. A periodic pattern of alternating vortices is shed from opposite sides of an obstacle in the incoming flow. (a) Re=20 (b) Re=200 (c) Re=2000.

to the impulse before performing the projection. This extra term represents the path integral of viscous and external forces along the trajectory of a virtual fluid particle, which can be seamlessly integrated into the marching process of the forward flow map. Note that

$$\boldsymbol{X} = \boldsymbol{\Psi}_{t,0}(\boldsymbol{x}), \boldsymbol{\Psi}_{t,\tau}(\boldsymbol{x}) = \boldsymbol{\Phi}_{0,\tau}(\boldsymbol{X}). \tag{7}$$

We select the midpoints as quadrature points for the path integral

$$\begin{aligned} &\frac{1}{\rho}\int_0^t \boldsymbol{\mathcal{F}}^T_{0,\tau}(\boldsymbol{X})\left(\mu\nabla^2\boldsymbol{u}_\tau + \boldsymbol{f}_\tau\right)(\boldsymbol{\Psi}_{t,\tau}(\boldsymbol{x}))\mathrm{d}\tau \\ &\approx \sum_{i=0}^{n-1}\frac{\Delta t}{\rho}\boldsymbol{\mathcal{F}}^T_{0,i+1/2}(\boldsymbol{X})\left(\mu\nabla^2\boldsymbol{u}_{i+1/2} + \boldsymbol{f}_{i+1/2}\right)(\boldsymbol{\Phi}_{0,i+1/2}(\boldsymbol{X})). \end{aligned} \tag{8}$$

During the calculation of midpoint velocities, we employ the central difference method to compute the viscous force. Additionally, the forward flow map is marched, and the path integral is performed. $\boldsymbol{\Phi}_{0,i+1/2}$ and $\boldsymbol{\mathcal{F}}^T_{0,i+1/2}$ were reused from the temporary variables in the RK4 Marching from $\boldsymbol{\Phi}_{0,i}$ to $\boldsymbol{\Phi}_{0,i+1}$, which do not lead to extra overhead. We accumulate the force terms to the initial velocity, which is subsequently multiplied by $\boldsymbol{\mathcal{T}}^T_{n,0}$ in the pullback operation.

5 Matrix-Free AMGPCG solver on GPU

5.1 Matrix-Free UAAMG

We number the multigrid levels from fine to coarse. The equation for the l th level is:

$$A_l x_l = b_l. \tag{9}$$

Starting with zero initial guess, smoothing yields a solution denoted as x_l^{old}. Smoothing effectively eliminates the high-frequency components of the error, while the low-frequency components are more

Algorithm 3 V-Cycle

Input: l, b_l
Output: x_l

$x_l \leftarrow 0$; ▷ start with zero initial guess
if $l ==$ nLevel $- 1$ **then** ▷ coarsest level
    multiple RBGS to $A_l x_l = b_l$;
    return x_l;
end if
one RBGS to $A_l x_l = b_l$; ▷ pre-smooth
$b_{l+1} \leftarrow R_l(b_l - A_l x_l)$ ▷ calculate residual and restrict
$x_{l+1} \leftarrow$ V-Cycle$(l + 1, b_{l+1})$
$x_l \leftarrow x_l + 2P_l x_{l+1}$ ▷ prolongate with scaling
one RBGS to $A_l x_l = b_l$; ▷ post-smooth
return x_l;

efficiently reduced at the coarse level. The residual for the current solution is restricted to the coarse level as the right-hand side:

$$A_{l+1}x_{l+1} = b_{l+1} = R_l(b_l - A_l x_l^{\text{old}}). \tag{10}$$

The coarse level solution is then prolonged to the fine level:

$$x_l^{\text{new}} = x_l^{\text{old}} + 2P_l x_{l+1}. \tag{11}$$

Since UAAMG is used as a preconditioner for the Conjugate Gradient method, scaling the prolongated value by 2 improves the convergence rate[Stüben 2001].

In algebraic multigrid, the matrices A_l and A_{l+1} satisfy an algebraic relationship determined by the restriction and prolongation operators, following the Galerkin principle:

$$A_{l+1} = R_l A_l P_l. \tag{12}$$

For the Poisson equation, the coefficient matrix at the finest level can be readily represented in matrix-free form, as each degree of freedom (DoF) only interacts with its neighboring DoF on the grid. When applying 8-to-1 restriction and constant prolongation, this property is preserved across the coarse levels. Consequently, the coefficient matrix at the coarse levels can also be expressed in matrix-free form, enabling coarsening to be performed in a fully matrix-free manner [Shao et al. 2022].

5.2 Data Structure

The variables and matrix coefficients are stored on a grid using a Structure of Arrays (SOA) layout. The matrix is represented by five data channels: one Boolean channel indicating whether a voxel corresponds to a degree of freedom (DoF), one floating-point channel storing the diagonal coefficient, and three floating-point channels representing the off-diagonal coefficients between a voxel and its neighboring voxels in the positive directions of the three axes.

The grid is partitioned into $8 \times 8 \times 8$ tiles. Within each tile, voxel data for each channel is stored contiguously in memory to enhance memory locality. The memory index of a voxel can be directly computed from its Cartesian coordinates and the grid dimensions using bitwise operations. The tile size is selected to align with the shared memory capacity of modern GPUs and ensure it is a multiple of the CUDA warp size.

5.3 Aggregated CUDA Kernels

The performance of AMGPCG is primarily constrained by memory access rather than computational workload. To enhance performance, both the volume and efficiency of memory access must be optimized. In a typical sequence of CUDA kernels, the output of one kernel often serves as the input to the next. By aggregating multiple CUDA kernels into a single kernel at critical stages, the number of read and write operations to global memory can be significantly reduced, thereby improving memory efficiency.

As shown in Algorithm 3, we employ a V-Cycle multigrid method with a Red-Black Gauss-Seidel (RBGS) smoother, where the smoother contributes to a significant portion of the computational time. Naively, executing one RBGS iteration requires two separate kernels: one for the red phase and another for the black phase. Additionally, during the downstroke, an extra kernel is needed to restrict the residual. By aggregating these kernels into a single unified kernel, we can effectively reduce the total memory access volume, thereby improving performance.

One challenge here is the data dependency between different stages. These dependencies can be described using the Manhattan distance on the grid. For two voxels $a_1 = (i_1, j_1, k_1)$ and $a_2 = (i_2, j_2, k_2)$, the Manhattan distance between them is defined as

$$M(a_1, a_2) = |i_1 - i_2| + |j_1 - j_2| + |k_1 - k_2|. \tag{13}$$

The Manhattan distance between a voxel a_1 and a tile b_2 is the minimum of the Manhattan distance between a_1 and the voxels within the tile b_2,

$$M(a_1, b_2) = \min_{a_2 \in b_2} M(a_1, a_2). \tag{14}$$

When a CUDA block calculates the residual and performs RBGS, it requires the x_l values not only for voxels within the corresponding tile but also for voxels whose Manhattan distance from this tile is 1. These dependent x_l values must be updated in the previous stage. To address this, we calculate the dependent x_l values on the fly during CUDA block execution and use shared memory to store intermediate results, reducing global memory accesses.

Our aggregated kernel for RBGS and restriction operates in four integrated stages. First, it computes the x_l values for red voxels whose Manhattan distance to the tile is less than 3, setting each value as the quotient of the right-hand side and the diagonal coefficient. Next, it performs Gauss-Seidel smoothing for black voxels whose Manhattan distance to the tile is less than 2. Afterward, the residual is calculated for voxels within the tile, and finally, the residual is restricted to the coarser level, with the updated x_l values written back. This integrated approach minimizes memory access overhead and maximizes computational efficiency. Similarly, prolongation and RBGS in the upstroke can be aggregated in an analogous manner, further optimizing performance.

In each iteration of the conjugate gradient method, two dot products are required. We refer readers to [Shewchuk 1994] for details on the preconditioned conjugate gradient algorithm. The element-wise summation in the dot product can be efficiently parallelized using the exclusive scan operation from NVIDIA's CUB library [Iannario et al. 2024], which can also be aggregated since the input arguments for each dot product are already available in the previous CUDA

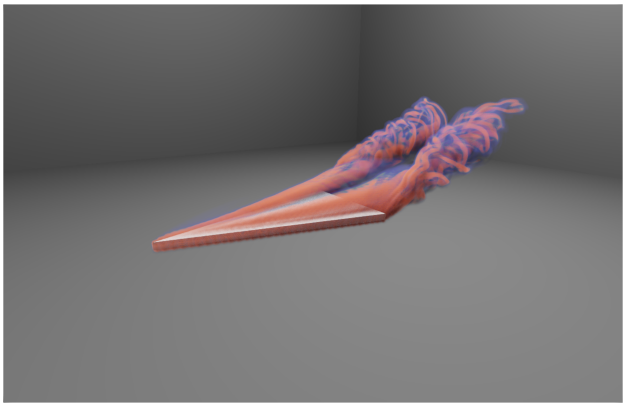


Fig. 9. Delta Wing. A delta wing subjected to an inflow at an attack angle of 20°. Vortical structures are generated behind the wings.

	NFM	LFM	Speedup
Marching	16.4 ms	4.8 ms	**3.4**×
Advection	-	0.9 ms	-
Pullback	2.5 ms	0.2 ms	**12.5**×
Error Correction	0.4 ms	0.03 ms	**13.3**×
Projection	102.5 ms	7.9 ms	**13.0**×
Flow Map Reset	0.7ms	0.06ms	**11.7**×
Operations for Smoke	-	0.2 ms	-
Total	122.5 ms	14.1 ms	**8.7**×

Table 3. Average time cost in one step of NFM and LFM in Leapfrog Vortex Rings. Both NFM and LFM reinitialize every 10 steps. Time cost for one step varies in a reinitalization cycle. Therefore, we report the average time cost.

kernel. In the previous CUDA kernel, each block can compute the element-wise product of the two arguments and perform a blockwise exclusive scan to sum the results. The summed result from each block is then written to a buffer. After the kernel execution completes, a global exclusive scan is applied to this buffer to compute the final result of the dot product. If a direct global multiplication and exclusive scan were used for the dot product, the process would involve reading one or two data channels for the multiplication, writing the intermediate result to a channel, and then reading it again for the exclusive scan. By aggregating the blockwise dot product computation into the previous CUDA kernel, we only need to read a single value for each tile during the global exclusive scan, which incurs negligible memory overhead. This optimization reduces the memory access volume by seven data channels per iteration of the conjugate gradient method, significantly improving performance.

5.4 Coefficients Trimming

In fluid simulation, only tiles adjacent to the boundary have non-uniform matrix coefficients. Before solving the equation, a tile is marked as trimmed if both the tile itself and all its neighboring tiles have uniform coefficients. The neighboring tiles are included in the evaluation because their coefficients are accessed during the RBGS smoothing process. For a trimmed tile, default coefficient values can be used, eliminating the need to read coefficients from global memory during the equation-solving process.

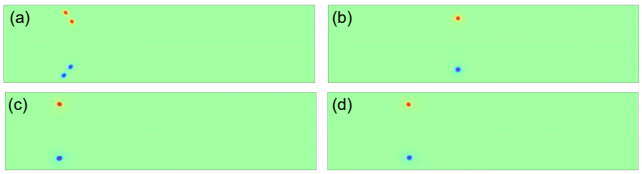


Fig. 10. 2D Leapfrog Marathon. The vortex pairs simulated by LFM remain separate after a round trip. (a) LFM (b) LFM without leapfrog method (c) PFM (d) NFM

Figure	Resolution	n	CG Iter	Runtime/Step	Total Runtime
14	256 × 128 × 128	10	15	14.1 ms	28.2 s
3	128 × 256 × 256	5	8	22.5 ms	22.5 s
5	384 × 256 × 256	5	8	70.0 ms	105.0 s
2	256 × 256 × 256	5	8	40.7 ms	203.5 s
4	128 × 256 × 128	4	6	11.7 ms	14.0 s
7	128 × 128 × 128	5	6	5.6 ms	10.0 s
9	256 × 128 × 128	5	6	11.1 ms	20.0 s

Table 4. Statistics: Here, n represents the number of steps in a reinitialization cycle. To avoid frequent interruptions of the CUDA stream for residual checks on the host side, we use a fixed number of iterations for the Conjugate Gradient method. For real-time examples, the reported runtime includes both simulation and rendering time. For non-real-time examples, the runtime reflects only the simulation time.

6 Experiments

In this section, we present the performance test results for LFM and our matrix-free AMGPCG solver on GPU, along with nine examples used to evaluate our method. Among these examples, Fire Ball, Trefoil, and Delta Wing are simulated and rendered in real time. The simulation results are passed to a ray-marching renderer implemented in Vulkan via CUDA-Vulkan interoperation. The remaining 3D examples are rendered using Houdini. All experiments were conducted on a desktop equipped with an NVIDIA RTX 4090.

6.1 Performance Analysis

6.1.1 LFM. We compare the time cost for NFM and LFM in Table 3. For this comparison, We tested the open-sourced implementation of [Deng et al. 2023], with the neural buffer replaced with direct memory storage to the global memory of GPU. Both NFM and LFM reinitialize every 10 steps. On average, our LFM implementation is 3.4× faster than the NFM implementation in marching, 13.0× faster in projection and 8.7× faster in total. The speed up is attributed to fewer advection, marching and projection in each reinitialization cycle and our fast matrix-free AMGPCG solver. In addition, while NFM requires pullback, error correction and flow map reset at every step, LFM performs these operations only during reinitialization. We also present the runtime of LFM for each example in Table 4.

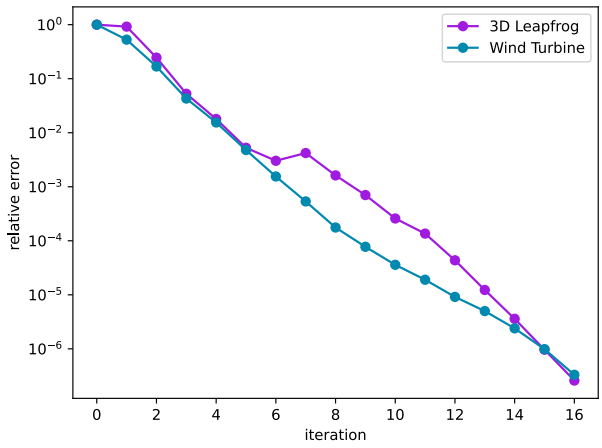


Fig. 11. Relative error of the linear solver for 3D Leapfrog and Wind Turbine.

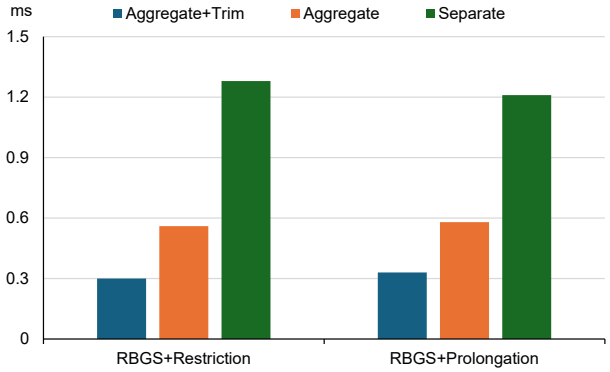


Fig. 12. Time cost on the finest level of multigrid

6.1.2 *AMGPCG solver.*

Convergence. While Geometric Multigrid relies on boundary smoothing to improve the convergence rate when solid obstacles appear in the domain [McAdams et al. 2010], Algebraic Multigrid does not encounter such difficulties due to its adherence to the Galerkin principle. In Figure 11, we illustrate the convergence of our AMGPCG solver for two 3D examples: Leapfrog and Wind Turbine. The former represents a uniform domain, while the latter involves a domain with a solid obstacle. The multigrid levels are selected such that the smallest dimension of the coarsest grid is 8. Additionally, we perform 10 RBGS smoothing steps on the coarsest level.

Aggregated kernels and data trimming. We evaluate the performance improvements from aggregated kernels and data trimming on a $256 \times 256 \times 256$ grid with a surrounding solid boundary (pure Neumann). Figure 12 illustrates the time cost for RBGS+Restriction and RBGS+Prolongation on the finest level. These operations dominate the computational time of a single V-Cycle, as the voxel number of each level forms a geometric sequence with a common ratio of $\frac{1}{8}$. Aggregated kernels gives 2× speed up compared to separate kernels, and data trimming gives a 2× speed up on top of that. Figure 13 presents the time breakdown for a single CG iteration. Because of the pure Neumann boundary, recentering is required in each CG iteration. Aggregated kernels and data trimming reduce the execution time of one V-Cycle from 2.78 ms to 0.81*ms*. Furthermore, aggregating the blockwise dot product into a preceding kernel effectively eliminates the time cost for the dot product. Since the operations preceding the dot product are memory-bound, the computational workload introduced by the blockwise exclusive scan does not add extra time. Additionally, the global exclusive scan, used to sum the results from each block, incurs minimal overhead.

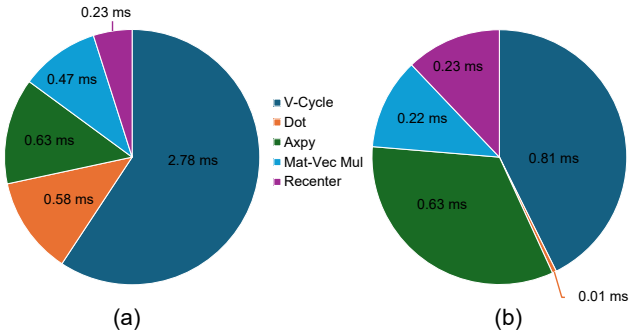


Fig. 13. Time breakdown of one CG iteration for a $256 \times 256 \times 256$ domain. (a) Separate kernels. (b) Aggregated kernels with data trimming.

Resolution		128^3	256^3	512^3
# Iterations	[Shao et al. 2022]	4	4	4
	Ours (Separate)	14	15	16
	Ours (Aggregate + Trim)	14	15	16
Build Time (ms)	[Shao et al. 2022]	15.4	69.1	556
	Ours (Separate)	0.073	0.493	3.44
	Ours (Aggregate + Trim)	0.153	0.906	6.78
Solve Time (ms)	[Shao et al. 2022]	51.2	510	4531
	Ours (Separate)	5.61	70.4	610
	Ours (Aggregate + Trim)	3.45	28.6	216

Table 5. We compare our solver with aggregated kernels and trimming, our solver with separate kernels and the SIMD UAAMG solver [Shao et al. 2022]. We report the number of iterations towards convergence, the time for building the multigrid hierachy and the time for solving the linear system. The convergence criterion is set to a relative error of 10^{-6} for all methods. The SIMD UAAMG solver [Shao et al. 2022] runs on a 13th Gen Intel(R) Core(TM) i9-13900F, and our solver runs on an Nvidia RTX 4090.

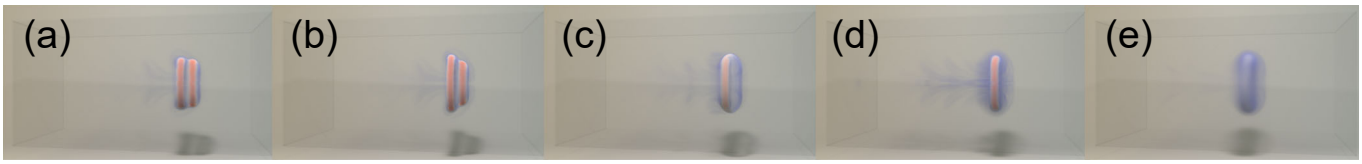


Fig. 14. Leapfrog Vortex Rings. The vortex rings simulated by LFM and NFM remain separated after 4 leaps. (a) LFM (b) NFM (c) PFM (d) Covector (e) BiMocq^2.

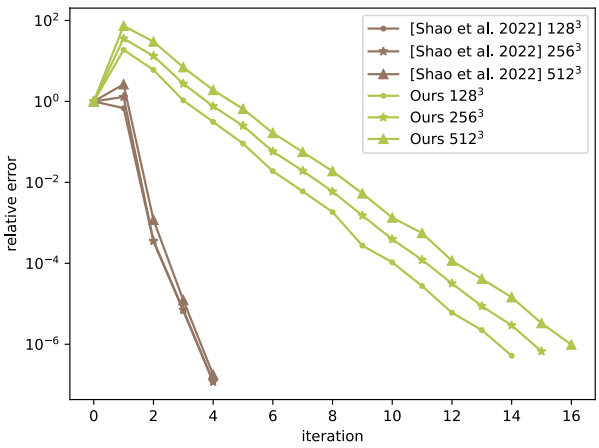


Fig. 15. Relative error of the SIMD UAAMG solver [Shao et al. 2022] and ours at different resolutions.

Comparison. We compare the convergence and performance of three configurations: our solver with aggregated kernels and data trimming, our solver without aggregated kernels and data trimming, and the SIMD UAAMG solver [Shao et al. 2022]. In the test scene, we set up cubic fluid tanks at different resolutions with Neumann boundary conditions. The right-hand side is constructed by adding uniform random noise (ranging from -1 to 0) to the normalized grid cell indices (ranging from 0 to 1), introducing both long-wavelength and short-wavelength errors. The convergence curves are plotted in Figure 15, and the performance statistics are shown in Table 5. Since aggregated kernels and trimming do not affect the convergence behavior of our solver, we present only one set of convergence curves for it. The SIMD UAAMG solver [Shao et al. 2022] achieves faster convergence than our solver, as it employs a W-cycle multigrid scheme with three pre-smoothings and post-smoothings, while our solver uses a V-cycle multigrid scheme with only one pre-smoothing and post-smoothing. Although the first scheme requires fewer iterations to converge, each iteration incurs more than three times the computational cost compared with the second scheme. Data trimming introduces some overhead during the construction of the multigrid hierarchy but accelerates the solve time. With aggregated kernels and data trimming, our solver achieves a **14.8**× to **21.0**× speedup in solving the linear system compared with the SIMD UAAMG solver [Shao et al. 2022].

6.2 Examples

2D Leapfrog. Figure 10 illustrates the classical 2D leapfrog test. Initially, two pairs of vortices are positioned on the left side of the domain. In a non-viscious setting, the vortex pairs should leap through each other and never merges. We compare LFM with NFM and PFM [Zhou et al. 2024] in this experiment. All methods reinitialize every 10 steps. After the vortex pairs reach the right boundary and bounce back, the vortex pairs simulated by NFM and PFM merged with each other. We also conduct an ablation study in this experiment. We replaced the leapfrog method in LFM with directly advecting the midpoint velocities sequentially. The modified method can not preserve the vortex pairs well.

Kármán Vortex Street. By incorporating viscosity through the path integral during forward marching, LFM is capable of simulating the Kármán vortex street phenomenon across various Reynolds numbers. The results are shown in Figure 8.

Leapfrog Vortex Rings. Figure 14 shows the leapfrogging behavior of two vortex rings. We compare LFM with NFM, PFM [Zhou et al. 2024], Covector Fluids [Nabizadeh et al. 2022] and BiMocq^2 [Qu et al. 2019]. All methods reinitialize every 10 steps. The vortex rings simulated by LFM and NFM merge after 5 leaps, 4 for PFM, 3 for Covector Fluids and 2 for BiMocq^2. These results demonstrate that our method achieves vortex preservation comparable to NFM.

Head-on Vortex Rings Collision. Figure 3 shows the head-on collision of two vortex rings. The two vortex rings initially have opposite vorticities and begin to approach each other. After collision, secondary vortex filaments are generated.

Vortex Rings Connection. Figure 5 shows that two vortex rings deform and connect with each other. A smaller vortex ring then separates off because of the fluctuation on the merged ring.

Wind Turbine. Figure 2 shows the vorticity field of a rotating wind turbine in an inlet. The wind turbine rotates with a constant angular velocity and a helical trail is created behind it.

Fire Ball. Figure 4 shows the real-time simulation and rendering results a fire ball. Our method captures the vortical flames generated by the uneven temperature distribution of the fire ball.

Trefoil. As shown in Figure 7, we simulate and render a trefoil knot in a real-time speed. The knot structure correctly breaks into one large vortex and another small vortex.

Delta Wing. Figure 9 illustrates the vorticity generated by a delta wing subjected to an inflow at an attack angle of 20°. This example is simulated and rendered in real time, allowing interactive adjustment of the attack angle by modifying the inflow direction.

7 Discussion and future work

In summary, we present Leapfrog Flow Maps (LFM), a hybrid velocity-impulse time integrator that achieves significantly lower computational workload compared to NFM while producing comparable results. Leveraging a highly optimized matrix-free Algebraic Multigrid Preconditioned Conjugate Gradient (AMGPCG) solver on GPU, our method is capable of simulating vortical phenomena in real time at a resolution of $256 \times 128 \times 128$ and simulating higher-resolution examples at a near-real-time rate.

Although our method is computationally efficient, its memory consumption can still be improved. Similar to NFM, LFM requires a buffer to store the velocity history for backward marching, which can lead to memory limitations for large simulation domains. We consider better quantization a promising technique for compressing the velocity buffer's memory overhead without substantially compromising performance in future work. In addition, integrating more complicated physics interactions, such as solid-fluid interaction with a particular focus on vortex-object interaction, will broaden the scope of the current method in accommodating real-time fluid applications.

Acknowledgements

We express our gratitude to the anonymous reviewers for their insightful feedback. Georgia Tech authors acknowledge NSF IIS #2433322, ECCS #2318814, CAREER #2420319, IIS #2433307, OISE #2433313, and CNS #1919647 for funding support. We credit the Houdini education license for video animations.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

References

Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Trans. Graph.* 36, 4 (2017).

Ryoichi Ando and Christopher Batty. 2020. A practical octree liquid simulator with adaptive surface resolution. *ACM Trans. Graph.* 39, 4 (2020).

Marsha J Berger and Joseph Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.* 53, 3 (1984).

Massimo Bernaschi, Pasqua D'Ambra, and Dario Pasquini. 2020. AMG based on compatible weighted matching for GPUs. *Parallel Comput.* 92 (2020), 102599.

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.* 22, 3 (2003).

Thomas F. Buttke. 1992. Lagrangian numerical methods which preserve the Hamiltonian structure of incompressible fluid flow. (1992).

Thomas F. Buttke and Alexandre J. Chorin. 1993. Turbulence calculations in magnetization variables. *Applied numerical mathematics* 12, 1-3 (1993), 47–54.

Duowen Chen, Zhiqi Li, Junwei Zhou, Fan Feng, Tao Du, and Bo Zhu. 2024. Solid-Fluid Interaction on Particle Flow Maps. *ACM Trans. Graph.* 43, 6 (2024).

Nuttapong Chentanez and Matthias Müller. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph.* 30, 4 (2011).

Ricardo Cortez. 1996. An impulse-based approximation of fluid motion due to boundary forces. *J. Comput. Phys.* 123, 2 (1996), 341–353.

Qiaodong Cui, Pradeep Sen, and Theodore Kim. 2018. Scalable laplacian eigenfluids. *ACM Trans. Graph.* 37, 4 (2018).

Yitong Deng, Hong-Xing Yu, Diyang Zhang, Jiajun Wu, and Bo Zhu. 2023. Fluid Simulation on Neural Flow Maps. *ACM Trans. Graph.* 42, 6 (2023).

Fan Feng, Jinyuan Liu, Shiying Xiong, Shuqi Yang, Yaorui Zhang, and Bo Zhu. 2023. Impulse Fluid Simulation. *IEEE Transactions on Visualization and Computer Graphics* 29, 6 (2023), 3081–3092.

Epic Games. 2024. *Unreal Engine.* version 5.5.

Ryan Goldade, Yipeng Wang, Mridul Aanjaneya, and Christopher Batty. 2019. An adaptive variational finite difference framework for efficient symmetric octree viscosity. *ACM Trans. Graph.* 38, 4 (2019).

Maria Iannario, Domenico Piccolo, and Rosaria Simone. 2024. *CUB: A Class of Mixture Models for Ordinal Data.* https://CRAN.R-project.org/package=CUB R package version 1.1.5.

Thomas Jakobsen. 2001. Advanced character physics. *In Game Developers Conference Proceedings* (2001).

JangaFX. 2024. *EmberGen.* version 1.2.

Michael Lentine, Wen Zheng, and Ronald Fedkiw. 2010. A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Trans. Graph.* 29, 4 (2010).

Wei Li, Yixin Chen, Mathieu Desbrun, Changxi Zheng, and Xiaopei Liu. 2020. Fast and scalable turbulent flow simulation with two-way coupling. *ACM Trans. Graph.* 39, 4 (2020).

Wei Li, Yihui Ma, Xiaopei Liu, and Mathieu Desbrun. 2022. Efficient kinetic simulation of two-phase flows. *ACM Trans. Graph.* 41, 4 (2022).

Zhiqi Li, Barnabás Börcsök, Duowen Chen, Yutong Sun, Bo Zhu, and Greg Turk. 2024a. Lagrangian Covector Fluid with Free Surface. In *ACM SIGGRAPH 2024 Conference Papers.* Article 43, 10 pages.

Zhiqi Li, Duowen Chen, Candong Lin, Jinyuan Liu, and Bo Zhu. 2024b. Particle-Laden Fluid on Flow Maps. *ACM Trans. Graph.* 43, 6 (2024).

Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Trans. Graph.* 35, 6 (2016).

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.* 23, 3 (2004).

Chaoyang Lyu, Kai Bai, Yiheng Wu, Mathieu Desbrun, Changxi Zheng, and Xiaopei Liu. 2023. Building a Virtual Weakly-Compressible Wind Tunnel Testing Facility. *ACM Trans. Graph.* 42, 4 (2023).

Chaoyang Lyu, Wei Li, Mathieu Desbrun, and Xiaopei Liu. 2021. Fast and versatile fluid-solid coupling for turbulent flow simulation. *ACM Trans. Graph.* 40, 6 (2021).

A. McAdams, E. Sifakis, and J. Teran. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* 65–74.

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3 (2013).

Mohammad Sina Nabizadeh, Ritoban Roy-Chowdhury, Hang Yin, Ravi Ramamoorthi, and Albert Chern. 2024. Fluid Implicit Particles on Coadjoint Orbits. *ACM Trans. Graph.* 43, 6 (2024).

Mohammad Sina Nabizadeh, Stephanie Wang, Ravi Ramamoorthi, and Albert Chern. 2022. Covector fluids. *ACM Trans. Graph.* 41, 4 (2022), 16 pages.

M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. 2015. AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM J. Sci. Comput.* (2015), S602–S626.

Valery Iustinovich Oseledets. 1989. On a new way of writing the Navier-Stokes equation. The Hamiltonian formalism. *Russ. Math. Surveys* 44 (1989), 210–211.

Ziyin Qu, Xinxin Zhang, Ming Gao, Chenfanfu Jiang, and Baoquan Chen. 2019. Efficient and conservative fluids using bidirectional mapping. *ACM Trans. Graph.* 38, 4 (2019).

Amir Hossein Rabbani, Jean-Philippe Guertin, Damien Rioux-Lavoie, Arnaud Schoentgen, Kaitai Tong, Alexandre Sirois-Vigneux, and Derek Nowrouzezahrai. 2022. Compact Poisson Filters for Fast Fluid Simulation. In *ACM SIGGRAPH 2022 Conference Proceedings (SIGGRAPH '22).* Article 35.

John W. Ruge and Klaus Stüben. 1987. Algebraic Multigrid. In *Multigrid Methods.* SIAM, 73–130.

Sergio Sancho, Jingwei Tang, Christopher Batty, and Vinicius C. Azevedo. 2024. The Impulse Particle-In-Cell Method. *Computer Graphics Forum* (2024), e15022.

Robert Saye. 2016. Interfacial gauge methods for incompressible fluid dynamics. *Science advances* 2, 6 (2016), e1501869.

Robert Saye. 2017. Implicit mesh discontinuous Galerkin methods and interfacial gauge methods for high-order accurate interface dynamics, with applications to surface tension dynamics, rigid body fluid–structure interaction, and free surface flow: Part I. *J. Comput. Phys.* 344 (2017), 647–682.

Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. 2005. A vortex particle method for smoke, water and explosions. *ACM Trans. Graph.* 24, 3 (2005).

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: a sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph.* 33, 6 (2014).

Han Shao, Libo Huang, and Dominik L. Michels. 2022. A fast unsmoothed aggregation algebraic multigrid framework for the large-scale simulation of incompressible flow. *ACM Trans. Graph.* 41, 4 (2022).

Jonathan R Shewchuk. 1994. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* Technical Report. USA.

Jos Stam. 1999a. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. 121–128.

Jos Stam. 1999b. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. 121–128.

Klaus Stüben. 2001. A review of algebraic multigrid. *J. Comput. Appl. Math.* 128, 1 (2001), 281–309. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

D.M. Summers. 2000. A Representation of Bounded Viscous Flow Based on Hodge Decomposition of Wall Impulse. *J. Comput. Phys.* 158, 1 (2000), 28–50.

Yuchen Sun, Linglai Chen, Weiyuan Zeng, Tao Du, Shiying Xiong, and Bo Zhu. 2024. An Impulse Ghost Fluid Method for Simulating Two-Phase Flows. *ACM Trans. Graph.* 43, 6 (2024).

Tetsuya Takahashi and Christopher Batty. 2023. A Multilevel Active-Set Preconditioner for Box-Constrained Pressure Poisson Solvers. *Proc. ACM Comput. Graph. Interact. Tech.* 6, 3 (2023).

Tetsuya Takahashi and Christopher Batty. 2025. A Primal-Dual Box-Constrained QP Pressure Poisson Solver With Topology-Aware Geometry-Inspired Aggregation AMG. *IEEE Transactions on Visualization and Computer Graphics* 31, 4 (2025).

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. 2017. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. 3424–3433.

E Weinan and Jian-Guo Liu. 1997. Finite difference schemes for incompressible flows in the velocity–impulse density formulation. *J. Comput. Phys.* 130, 1 (1997), 67–76.

E Weinan and Jian-Guo Liu. 2003. Gauge method for viscous incompressible flows. *Communications in Mathematical Sciences* 1, 2 (2003), 317–332.

Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast Fluid Simulations with Sparse Volumes on the GPU. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018)* 37, 2 (2018).

Cheng Yang, Xubo Yang, and Xiangyun Xiao. 2016. Data-driven projection method in fluid simulation. *Comput. Animat. Virtual Worlds* 27, 3–4 (2016).

Shuqi Yang, Shiying Xiong, Yaorui Zhang, Fan Feng, Jinyuan Liu, and Bo Zhu. 2021. Clebsch gauge fluid. *ACM Trans. Graph.* 40, 4 (2021).

Omar Zarifi. 2020. Sparse Smoke Simulations in Houdini. In *ACM SIGGRAPH 2020 Talks*. Article 3.

Junwei Zhou, Duowen Chen, Molin Deng, Yitong Deng, Yuchen Sun, Sinan Wang, Shiying Xiong, and Bo Zhu. 2024. Eulerian-Lagrangian Fluid Simulation on Particle Flow Maps. *ACM Trans. Graph.* 43, 4 (2024).