GPU Optimization for High-Quality Kinetic Fluid Simulation

Yixin Chen, Wei Li, Rui Fan, and Xiaopei Liu

Abstract—Fluid simulations are often performed using the incompressible Navier-Stokes equations (INSE), leading to sparse linear systems which are difficult to solve efficiently in parallel. Recently, kinetic methods based on the adaptive-central-moment multiple-relaxation-time (ACM-MRT) model [1], [2] have demonstrated impressive capabilities to simulate both laminar and turbulent flows, with quality matching or surpassing that of state-of-the-art INSE solvers. Furthermore, due to its local formulation, this method presents the opportunity for highly scalable implementations on parallel systems such as GPUs. However, an efficient ACM-MRT-based kinetic solver needs to overcome a number of computational challenges, especially when dealing with complex solids inside the fluid domain. In this article, we present multiple novel GPU optimization techniques to efficiently implement high-quality ACM-MRT-based kinetic fluid simulations in domains containing complex solids. Our techniques include a new communication-efficient data layout, a load-balanced immersed-boundary method, a multi-kernel launch method using a simplified formulation of ACM-MRT calculations to enable greater parallelism, and the integration of these techniques into a parametric cost model to enable automated prameter search to achieve optimal execution performance. We also extended our method to multi-GPU systems to enable large-scale simulations. To demonstrate the state-of-the-art performance and high visual quality of our solver, we present extensive experimental results and comparisons to other solvers.

Index Terms—GPU optimization, parallel computing, fluid simulation, lattice Boltzmann method, immersed boundary method

1 Introduction

Fluid simulations in computer graphics mostly rely on solving the incompressible Navier-Stokes equations (INSE) using certain types of discretization schemes [3], [4], [5], [6], [7]. While there are a vast number of available solvers for INSE, including some which achieve relatively high accuracy for turbulent flows (e.g., [7], [8]), most methods require solving large global sparse linear systems, such as the pressure equation or equations resulting from implicit formulations for stability, making these solvers difficult to efficiently parallelize, especially on GPUs or for high-resolution simulations. For this reason, local but accurate solvers involving no global equations are desirable.

Over the years, kinetic solvers based on the lattice Boltzmann equations (LBE) [9] have been considered as an appealing alternative for high-performance fluid simulations. Unlike many INSE solvers, kinetic solvers only rely on local schemes, and thus by nature are well suited to parallelization. These methods are also conservative by construction, making them theoretically ideal for turbulent flow simulations. However, despite previous attempts [10], [11], [12], [13], kinetic solvers have seldomly been used in computer graphics applications due to their poor accuracy and stability. One reason is that most previous works relied on the BGK [14] or raw-moment multiple-relaxation-time (RM-MRT) [15], [16] models, which are known to exhibit noticeable visual artifacts. The recent development of the adaptive-central-moment multiple-relaxation-time (ACM-MRT) model [1], [2] for kinetic solvers has helped overcome these limitations, leading to an accurate and stable solver demonstrating equal and sometimes superior visual quality compared to state-of-the-art INSE solvers (see Fig. 1), while also offering the potential for high computational efficiency. In addition, ACM-MRT-based kinetic solvers can be easily combined with the immersed boundary (IB) method [17] to enable flexible handling of boundaries with complex geometries for both static and moving solids immersed in the fluid domain [2].

Nevertheless, despite the advancements offered by the ACM-MRT model, a straightforward GPU implementation of it does not produce high performance. There have been multiple attempts in the literature to improve the parallelism of GPU-based kinetic solvers, mostly relying on optimizations to data layout [18], [19] or the order of computations [20]. However, these works targeted the traditional BGK [21] and RM-MRT models [15], which typically cannot attain the visual quality needed for graphical fluid simulations. Furthermore, to the best of our knowledge, there is no work on GPU optimizations for enhancing the performance of IB-based kinetic solvers to support the immersion of complex dynamic solids in fluids.

In this paper, we aim to increase the performance of kinetic solvers using the ACM-MRT model and IB method on GPUs by addressing several key issues. First, when using the IB method in a kinetic solver, the data layout should be designed to maximize the correspondence between spatial locality in

• *The authors are with the School of Information Science and Technology, Shanghai Engineering Research Center of Intelligent Vision and Imaging, ShanghaiTech University, Shanghai 201210, China.*
E-mail: {chenyx2, liwei, fanrui, liuxp}@shanghaitech.edu.cn.

Manuscript received 19 January 2020; revised 28 January 2021; accepted 3 February 2021. Date of publication 16 February 2021; date of current version 1 August 2022.
(Corresponding author: Xiaopei Liu.)
Recommended for acceptance by X. M Tricoche.
Digital Object Identifier no. 10.1109/TVCG.2021.3059753

Fig. 1. *Simulation of turbulent smoke passing through complex architecture*. With our GPU optimized kinetic solver on a multi-GPU system, we can efficiently simulate this high-resolution scenario (grid resolution $1200 \times 250 \times 840$, with 3,688,157 solid samples) at a rate of 1.21 seconds per time step, taking 76,600 time steps in total to produce an animation of 10 seconds. Here wind moves from left to right through the buildings and smoke is continuously injected from the left and advected by the wind. Comparably high computational efficiency and visual fidelity is very difficult to achieve using parallel Navier-Stokes solvers at such high grid resolutions.

the simulation domain and locality in GPU memory. For this purpose, we propose a new parametric data layout model which seeks to maximize data access contiguity. Second, a straightforward parallel implementation of the IB method may introduce substantial load imbalance due to large variations in the number of solid samples around different fluid nodes. We propose a new, more efficient method to implement IB which parallelizes over solid samples instead of fluid nodes to significantly improve load balance. Finally, the ACM-MRT model performs lengthy calculations which require a large number of registers for each GPU thread, leading to reduced occupancy and execution efficiency. We perform multiple kernel launches and simplify the ACM-MRT mathematical expressions to mitigate these issues. These multifaceted optimizations are integrated into a single parametric cost model, allowing us to perform an automated search in the parameter space to obtain the best execution performance. We also extend our techniques to multi-GPU systems and show that our solver can easily and efficiently support high-resolution large-scale simulations.

In summary, we make the following key technical contributions to high-performance fluid simulations using IB-based kinetic solvers with the ACM-MRT model:

- A new parametric data layout to improve memory performance for IB-based kinetic solvers, improving access contiguity for both fluid and solid data.
- An optimized parallel IB implementation with separate domain boundary treatments to improve load balancing and reduce warp divergence.
- Tuning the number of kernel launches and simplifying the mathematical formulation of the ACM-MRT model to improve GPU occupancy and execution efficiency.
- An integrated cost model with automated parameter search to produce the best settings for each simulation instance.

With the above optimizations, our new implementation of a kinetic solver for the ACM-MRT model can run 5 to 10 times faster than a baseline GPU implementation. It also significantly outperforms state-of-the-art GPU-based INSE solvers, being about an order of magnitude faster at the same grid resolution and similar visual quality. Due to its faster rate of convergence, our solver can even produce visual quality similar to INSE solvers while using a significantly lower grid resolution, which leads to a two order of magnitude improvement in performance. High-resolution fluid simulations on multi-GPU systems can also be efficiently implemented. Fig. 1 shows an example of a large-scale multi-GPU simulation at a resolution of $1200 \times 250 \times 840$. We also present a number of performance comparisons, analyses and visual animations to demonstrate the capabilities of our solver to produce high quality, robust fluid simulations at practically useful speeds.

2 Related Work

Our work is related to high-performance fluid simulations on the GPU. We first review different fluid simulation methods for both INSE and kinetic solvers, and then review different optimization and parallelization techniques.

2.1 Fluid Simulations

2.1.1 Incompressible Navier-Stokes Solvers

There has been a vast number of INSE solvers developed in the computer graphics community. Early works stemmed from Stam [22] and used stable semi-Lagrangian advection, but suffered from excessive numerical diffusion. This issue was progressively improved using higher-order schemes [7], [8], [23], [24], [25], [26]. Another way to counteract numerical diffusion is to use vortex-based methods [27], [28], [29], [30], [31], [32], [33], [34], where the velocity field is corrected by vorticity distributions. Noise-based methods [35], [36] are a cheap way to preserve flow details, but the resulting simulation may be non-physical and appear unrealistic. These approaches have recently been improved using neural network-based methods [37], [38], [39], [40]. While the above works employed only uniform grids, non-uniform grids [4], [41], [42] or even tetrahedral meshes [3], [43], [44] have been used to perform adaptive computations.

In addition to grids, INSE can be solved using particle methods [5], [45], [46], [47], [48], with more recent techniques allowing for volume conservation [49], improved incompressibility [50] and better boundary treatments [51], [52]. To retain the benefits of both grids and particles for better advection and incompressibility, hybrid methods combining

particles and grids have been developed [6], [33], [53], [54], [55]. Note that particle-based methods without an auxiliary grid may not need global sparse linear system solvers, but they usually have difficulty simulating turbulent flows unless a very large number of particles is used.

2.1.2 Kinetic Solvers

Instead of solving INSE directly, kinetic methods solve the more general Boltzmann equation, which approximates INSE with different degrees of accuracy depending on how relaxations in collisions is modeled. Early kinetic solvers used the lattice BGK (or single-relaxation-time) model [14]. To improve accuracy and stability, raw-moment multiple-relaxation-time (RM-MRT) model [15] was developed, but it had problems with turbulent flows due to violation of Galilean invariance, and leads to strong dispersion errors at high Reynolds numbers. Recent improvements in relaxation modeling include the cascaded relaxation [56], [57] and central-moment multiple-relaxation-time (CM-MRT) models [58], [59], especially the non-orthogonal version [60], which much better respects Galilean invariance. However, these works still failed to address how to determine high-order relaxation parameters, which has been found important for turbulent flow simulations. Based on observations and analyses of the non-orthogonal CM-MRT model, Li *et al.* [1], [2] proposed adaptive schemes for relaxation parameters based on local flow characteristics. This model, called ACM-MRT, allows for much more accurate simulations of both laminar and turbulent flows.

When solids are immersed inside the fluid domain, boundary conditions need to be satisfied. Bounce-back schemes have been used to enforce no-slip [61] or slipping [62] conditions. To improve accuracy, curved boundary conditions were developed [63], which can also be extended to support dynamic solids [64]. However, these methods may encounter serious problems for turbulent flows when grid resolution around the solid boundary is not sufficiently high. A more flexible technique for boundary treatment is based on the immersed boundary method [17], [65], [66], which can be used to handle both static and dynamic solid boundaries using penalty forces.

2.2 Parallel Performance Optimizations

2.2.1 Parallel Navier-Stokes Solvers

As mentioned earlier, most INSE solvers require solving large global equations and are therefore difficult to parallelize. Parallel INSE solvers for multicore systems were developed using OpenMP [67] and extended to multi-node cluster systems using MPI [68]. In computer graphics, Mashayekhi *et al.* [69] proposed a system called "Nimbus", which automatically distributes grid-based and hybrid simulations across cloud computing nodes for faster execution at higher grid or particle resolutions.

INSE solvers can also be parallelized on the GPU. Early works tried to map the entire computation into texture memory and use fragment programs to solve INSE [70], [71], [72]. With the advent of CUDA [73], optimized parallel sparse linear solvers became available [74] on single and multi-GPU systems, making GPU implementations of many INSE solvers much simpler [75], [76], [77], [78].

Recently, GPU optimizations for the parallel material-point method (MPM) were proposed to accelerate a hybrid fluid solver [79]. In addition, an efficient large-scale fluid simulator on the GPU using the fluid-implicit particle (FLIP) method [80] and a high-level, data-oriented programming language for sparse data structures (Taichi) [81] has been proposed. To utilize the computational advantages of both CPUs and GPUs, some works proposed parallel INSE solvers on hybrid heterogeneous systems [82], [83], [84].

2.2.2 GPU Optimizations for Parallel Kinetic Solvers

While INSE solvers are inherently difficult to parallelize, kinetic solvers use a local formulation and thus are naturally scalable and well suited for implementation on highly parallel GPUs. Similar to earlier parallelization methods for INSE solvers on the GPU, texture and render buffers were initially used to accelerate kinetic solvers [85], [86]. Using CUDA, GPU implementations of kinetic solvers became much simpler. However, a straightforward implementation of a kinetic solver does not achieve very good performance. One issue is memory coalescing, which is crucial for high GPU performance. The array-of-structures (AoS) data layout, which performs well on CPUs [87], was replaced by a structure-of-arrays (SoA) data layout on the GPU [20]. The collected SoA (CSoA) data layout, which combines AoS and SoA, was recently proposed [19] to further enhance performance and allow better caching when solid boundaries are complex or domain boundaries are irregular [18].

To reduce non-coalesced memory accesses, the use of a single kernel launch with swapping for both the streaming and collision stages of a simulation has been proposed [88]. Since non-coalesced reads require less time than non-coalesced writes, the pull-in scheme was argued to be better than the push-out scheme [89]. With the above optimizations, kinetic solvers using different lattice models (D2Q9 [90], D3Q13 [20], D3Q19 [91], [92]) were proposed on the GPU, based mostly on the BGK or RM-MRT collision models. In addition, to simulate larger scenarios, multi-GPU kinetic solvers were proposed using CUDA [93], [94] and OpenMP [95]. To our knowledge, there has been no work on GPU optimizations for kinetic solvers with the D3Q27 lattice for either the BGK or RM-MRT collision models.

Our Contributions. Most of the GPU optimizations for kinetic solvers described above were for fluids only. When complex solids are present in fluids and boundary conditions are enforced using the immersed boundary method, the previous techniques may lead to poor performance due to sub-optimal data layout, load imbalance and warp divergence. Furthermore, when the ACM-MRT model is used to achieve high visual quality, a straightforward implementation may result in low GPU occupancy and further depress performance. This paper addresses these issues and presents a simulator which is substantially faster than state-of-the-art GPU-based INSE solvers, while at the same time producing equal or higher visual quality.

3 Computational Framework

Before introducing our GPU optimizations in detail, we first briefly review the computational framework employed for fluid simulations in this paper. Fluid flows are usually

modeled using the Navier-Stokes (NS) equations, but can alternatively be described by the Boltzmann equation (BE) [96], which is more general and can recover the NS equations under certain constraints [9]. BE can usually be solved by a set of lattice Boltzmann equations (LBE) [9], which are local and conservative with respect to density and momentum:

$$f_i(\boldsymbol{x}_k + \boldsymbol{c}_i, t+1) - f_i(\boldsymbol{x}_k, t) = \Omega_i(\rho, \boldsymbol{u}) + G_i(\boldsymbol{g}). \tag{1}$$

Here, k is the position index of each grid node; $t \in \{1, 2, \ldots\}$ is the time step index, and $i \in \{0, 1, \ldots, 26\}$ is the index of discrete particle velocities $\boldsymbol{c}_i$ used in the D3Q27 lattice structure[1] of the ACM-MRT model. Each $\boldsymbol{c}_i$ is associated with a corresponding discretized distribution function f_i, and $\boldsymbol{x}_k + \boldsymbol{c}_i$ is exactly located at a nearby node. Finally, Ω_i and G_i are the discretized collision and external force terms, and the macroscopic density ρ and velocity $\boldsymbol{u}$ can be obtained by taking the following moments w.r.t $\boldsymbol{c}_i$:

$$\rho = \sum_i f_i, \qquad \boldsymbol{u} = \frac{1}{\rho}\sum_i \boldsymbol{c}_i f_i. \tag{2}$$

Due to its simple algebraic formulation, the solution procedure for LBE can usually be split into the following three steps, starting with *streaming*:

$$f_i^*(\boldsymbol{x}, t) = f_i(\boldsymbol{x} - \boldsymbol{c}_i, t). \tag{3}$$

Then, macroscopic quantities (ρ and $\boldsymbol{u}$) are computed (Eq. (2)) and boundary treatment is applied. Finally, we perform the *collision* step with an external force term G_i:

$$f_i(\boldsymbol{x}, t+1) = f_i^*(\boldsymbol{x}, t) + \Omega_i(\rho^*, \boldsymbol{u}^*) + G_i, \tag{4}$$

where ρ^* and $\boldsymbol{u}^*$ are computed by the streamed distribution functions f_i^* after applying boundary treatments.

Key to ensuring accurate and stable fluid simulations for kinetic solvers is the formulation of Ω_i. Here, we employ the ACM-MRT model [1], [2], which is constructed by a relaxation process with multiple rates:

$$\boldsymbol{\Omega} = -M^{-1}DM(\boldsymbol{f} - \boldsymbol{f}^{eq}) = -M^{-1}D(\boldsymbol{m} - \boldsymbol{m}^{eq}), \tag{5}$$

where for each grid node, $\boldsymbol{f}$, $\boldsymbol{f}^{eq}$, $\boldsymbol{m}$, $\boldsymbol{m}^{eq}$ and $\boldsymbol{\Omega}$ are vectors containing the assembly of distribution functions f_i, their equilibrium f_i^{eq}, their corresponding distributions in moment space m_i and m_i^{eq}, and the collision results Ω_i, respectively. M is a 27×27 moment-space projection matrix, and D is a diagonal matrix with the same dimensions containing different relaxation rates. Low-order relaxation rates are determined by physical parameters (e.g., kinematic viscosity ν), while high-order relaxation rates should be determined adaptively, balancing local dissipation and dispersion errors (see [1] and *supplementary document, available online*, in [2]).

1. See Fig. 2 for an illustration, and the *supplementary document, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TAFFC.2021.3059688*, for the specific values of $\boldsymbol{c}_i$.

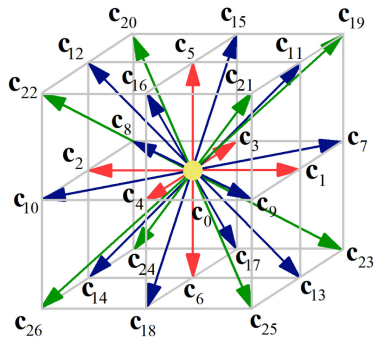


Fig. 2. *Lattice structure.* D3Q27 lattice structure, where $\boldsymbol{c}_i$'s are discretized microscopic velocities associated with the corresponding f_i.

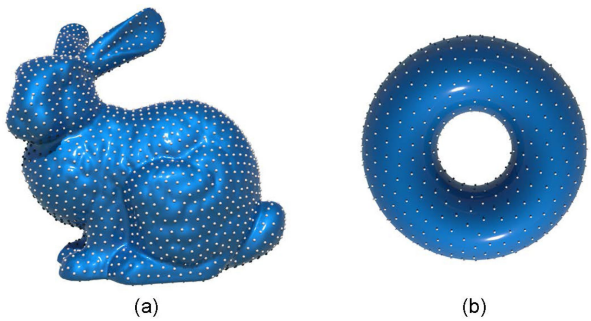
3.1 Boundary Treatment

When solids are immersed inside fluids, boundary conditions need to be satisfied. The traditional method for boundary treatment in kinetic solvers is to use bounce-back schemes [12], [97]. However, when dealing with curved or moving solids, these schemes are either inaccurate or inflexible. An alternative method which balances flexibility and accuracy for boundary treatment is the immersed boundary (IB) method [2], [17], [65], [66] we employ in this paper, which usually follows three main steps:

- *Solid surface sampling*. First, we uniformly sample the solid surface using Poisson-disk surface sampling [98], see Fig. 3. The sampling should be dense enough such that each fluid cell around the solid boundary contains 10 to 100 samples (depending on the local geometry of the solid) to ensure sufficient accuracy.
- *Force computation at solid samples*. A solid sample $\boldsymbol{x}_s$ may not be located exactly at a fluid grid node (see Fig. 4 for a 2D illustration). Thus we first interpolate fluid velocities $\boldsymbol{u}(\boldsymbol{x}_s)$ from nearby fluid nodes (cf. the green box in Fig. 4) using a kernel function $K(\cdot)$:

$$\boldsymbol{u}(\boldsymbol{x}_s) = \sum_{i \in N(\boldsymbol{x}_s)} K(\|\boldsymbol{x}_s - \boldsymbol{x}_f^i\|)\boldsymbol{u}(\boldsymbol{x}_f^i). \tag{6}$$

  $\boldsymbol{u}(\boldsymbol{x}_s)$ may be different from the desired boundary velocity $\boldsymbol{u}_b(\boldsymbol{x}_s)$, so a penalty force should be applied to the fluid at the solid sample location $\boldsymbol{x}_s$:



(7)

Fig. 3. *Surface sampling used for the immersed boundary method.* We employ Poisson-disk sampling to generate uniformly distributed solid samples over the surface, which are used to enforce boundary conditions at the solid boundary. Note that the samples here are for illustration only; in practice the sampling should be much denser.

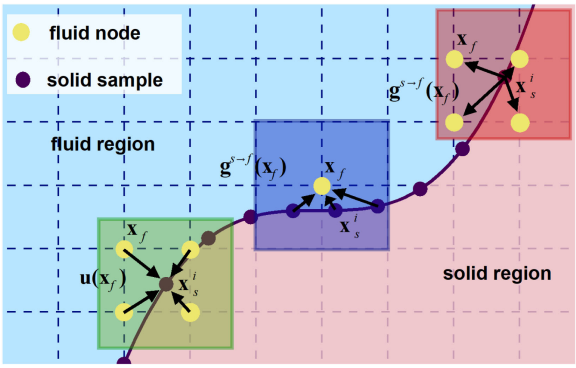


Fig. 4. *Implementing the immersed boundary method.* We illustrate two ways to compute the penalty force at each fluid node bordering the solid boundary. The fluid velocity is first interpolated from the neighboring fluid nodes (green box). Then to compute the penalty force at each fluid node: 1) boundary samples are first found in a region around the fluid node, possibly using a grid accelerated data structure, then the forces are summed with a spreading kernel to form the force at the fluid node (blue box); or 2) starting from boundary samples, their penalty forces are added directly to the nearby fluid nodes (red box).

- *Force spreading to fluid nodes.* Since $\boldsymbol{g}^{s\to f}(\boldsymbol{x}_s)$ is applied at the solid sample, we need to spread it to the nearby fluid nodes $\boldsymbol{x}_f$ using the same kernel:

$$\boldsymbol{g}^{s\to f}(\boldsymbol{x}_f) = \sum_{i\in N(\boldsymbol{x}_f)} K(\|\boldsymbol{x}_s^i - \boldsymbol{x}_f\|)\boldsymbol{g}^{s\to f}(\boldsymbol{x}_s^i). \tag{8}$$

  We use the smallest $2 \times 2 \times 2$ kernel in 3D (and 2×2 kernel in 2D) to preserve vortices around solids [99].

Note that the number of solid samples involved in the spreading process depends on the locations of the samples, and can vary significantly over the solid surface. In addition, with the IB method, solids are allowed to move inside fluids, creating turbulent flows at high speeds or with low viscosity. See the supplementary document, available online, for more details of the IB method used in the kinetic solver.

3.2 A Baseline GPU Implementation

With the procedures described above, we can produce a non-optimized baseline GPU implementation of an IB-based kinetic solver with the ACM-MRT model as follows:

- *Initialization.* Before starting the simulation, we allocate global memory on the GPU for the f_i values at times t and $t+1$, and for the density ρ, velocity $\boldsymbol{u}$ and external force $\boldsymbol{g}$ at time t, for each fluid node in the simulation. Vector fields (e.g., $\boldsymbol{f}$ and $\boldsymbol{u}$) are stored as linear arrays of vectors.
- *Streaming.* At each simulation time step t, we parallelize over the fluid nodes, and advect f_i from the neighbors of each fluid node (cf. Eq. (3)) by reading and writing to GPU global memory.
- *Calculating macroscopic fields.* After streaming, we again parallelize over the fluid nodes and compute the density ρ and velocity $\boldsymbol{u}$ at each node (cf. Eq. (2)).
- *Boundary treatment.* We employ the IB method for boundary treatment. When computing penalty forces at solid samples (cf. Eqs. (6) and (7)), we parallelize over the sample points and access the nearby fluid nodes (cf. the green box in Fig. 4). However, when spreading forces (cf. Eq. (8)), we parallelize over the fluid nodes in the bounding box(es) of the solid object (s) (cf. the blue box in Fig. 4) and access the nearby solid samples. Note that this parallelization procedure directly follows the order of the equations defining the IB method.
- *Collision.* After boundary treatment, we perform collisions using Ω_i and the force term G_i (cf. Eq. (4)), which is done by parallelizing over the fluid nodes. We perform all collision computations in a single kernel, in keeping with previous GPU-based lattice Boltzmann implementations.
- *Update.* Finally, with Ω_i and G_i computed, we update f_i and move to time step $t+1$.

Lastly, note that the matrix-based computations during the collision step (cf. Eq. (5)) are typically implemented by fully expanding the computation into arithmetic operations between scalar values, rather than through calls to matrix multiplication subroutines.

4 High-Performance GPU Optimizations

The baseline GPU implementation described in Section 3.2 is expected to be highly scalable due to the local formulation of the kinetic solver. However, as we discussed in Section 1, a number of optimizations can be performed to further improve its speed by more than an order of magnitude in certain simulation scenarios. In this section, we present a number of optimization techniques, and also show how to integrate these techniques into an automated optimal parameter searching framework to enable fast fluid simulations.

4.1 GPU Thread Assignment

Before introducing our GPU optimizations in detail, we first describe how threads are usually assigned in GPU-based kinetic fluid simulations containing immersed solids. Depending on which part of the simulation is being performed, a GPU thread can be assigned to perform the work at either a fluid node or a solid sample. As fluid nodes are typically laid out in a 3D grid, we use 3D thread blocks to index threads assigned to the fluid nodes. We ensure that the size of the first dimension of the thread block is a multiple of the GPU warp size (typically 32), while the sizes of the other dimensions are user tunable and depend on the type of GPU being used. Solid samples, while used to describe a 3D surface, are typically specified using a 1D index. Therefore, we also use 1D indices, possibly with a permuted ordering, for threads assigned to solid samples.

4.2 Improving Memory Performance

A number of previous works on GPU optimized kinetic solvers [18], [100], [101] have pointed out the importance of data layout on simulation performance. GPU main memory is laid out in *segments* of consecutive addresses, typically 128 bytes in size. GPU threads execute in units called *warps*, and a warp of threads accessing data from a single memory segment achieves much higher performance than the warp accessing data from multiple segments. Thus, an important

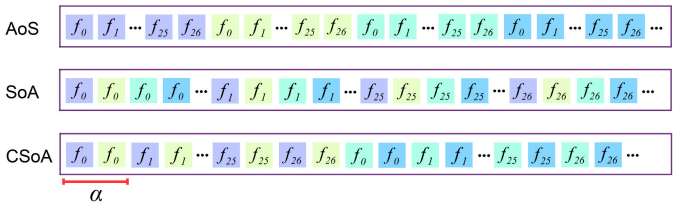


Fig. 5. *Different data layouts of fluid nodes in memory.* AoS: All f_i values from a node are stored consecutively, followed by f_i values of subsequent nodes. SoA: For each $i \in \{0, \ldots, 26\}$, f_i values for all the nodes are stored consecutively. CSoA: For each $i \in \{0, \ldots, 26\}$, the f_i values of α nodes, for some choice of α, are stored consecutively.

principle in GPU memory optimization is *memory coalescing*, i.e., ensuring a warp of threads access contiguous, segment-aligned memory locations.

4.2.1 Optimal Data Layout for Fluid Nodes

We first look at how data for fluid nodes should be laid out in GPU memory. Fluid data can be described using a number of vector fields, e.g., for different f_i's, $\boldsymbol{u}$ and $\boldsymbol{g}$. Collections of vectors are usually stored in one of the two ways, as an array-of-structures (AoS) [87], where each element in an array is a structure storing all vector components of a fluid node (cf. Fig. 5 (top) for an AoS layout of the f_i vectors), or as a structure-of-arrays (SoA), where each vector component of the fluid nodes over the entire domain is first grouped together and then stored consecutively in an array (cf. Fig. 5 (middle)). Recall that we assign one thread to process each fluid node. During the streaming step of the simulation, threads need to access f_i values from neighboring nodes. To coalesce these memory accesses, many works adopt the SoA layout for fluid data [20], [93]. Recent works have proposed the CSoA data layout [18], [19] to further improve coalescing and caching effects. Let α denote the number of fluid nodes that we collect into a group (cf. Fig. 5 (bottom)) and β the number of components that each fluid node vector contains (e.g., $\beta = 27$ for $\boldsymbol{f}$ in the D3Q27 model). Then the i'th component of the vector for the k'th fluid node is stored at the following memory location:

$$\beta\alpha\lfloor\frac{k}{\alpha}\rfloor + \alpha i + k \bmod \alpha. \tag{9}$$

4.2.2 Optimal Layout for the Immersed Boundary Method

When solid objects are immersed inside the fluid domain, boundary conditions need to be satisfied at the solid surface. As stated in Section 1, we use the IB method to enforce boundary conditions instead of the traditional bounce-back scheme. This is because the bounce-back scheme (as shown in Fig. 6 and Algorithm 1) requires each grid location to be labeled as either solid or fluid, leading to conditional identification of the distribution functions and warp divergence, which slows down the simulation. The IB method on the other hand only uses penalty forces around solid boundaries, which allows overlap of solid and fluid regions and does not require conditional identification of solid nodes, leading to better parallel performance.

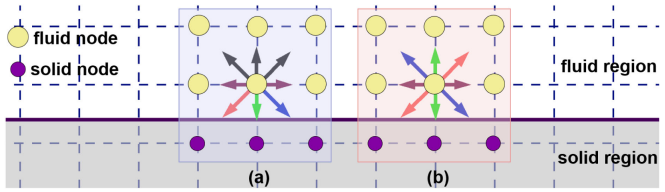


Fig. 6. *Bounce-back scheme for a non-slip boundary.* During streaming at each fluid node, some nearby grid nodes may be located inside the solid region (e.g., the purple nodes in (a) and (b)), and the distribution functions (gray arrows) are unknown and need to be reconstructed. The bounce-back scheme uses the corresponding distribution functions in the opposite directions (arrows with the same color in (b)) to fill in these unknown distribution functions, ensuring the no-slip condition. Since the solid boundary location around a fluid node may be arbitrary, conditional statements are needed in the kernel code to identify which set of nearby grid nodes are inside solid regions and require bounce-back treatment. Since different kernel threads may follow different conditional branches, warp divergence may occur to slow down the simulation.

Algorithm 1. Bounce-Back Scheme in Kinetic Solver

```
if a grid node at x is fluid node then
    for each direction c_i of the node do
        if the grid node at x − c_i is solid node then
            set f_i*(x, t) = f_i'*(x, t); ▷ f_i'* is opposite to f_i*.
        end if
    end for
end if ▷ Bounce-back scheme on f_i* after streaming.
```

The main problem we encountered when dealing with solids is that solid samples may initially be stored in an arbitrary order in GPU memory, depending on how the solid surface was triangulated or how samples within each triangle were ordered. Recall from Section 3 that velocity is interpolated at each solid sample from nearby fluid nodes and penalty forces are spread to fluid nodes from solid samples. Computing these quantities requires solid samples to access nearby fluid data, and hence the order of the solid samples affects the memory access pattern and performance. Note that this behavior has not been previously discussed in the literature.

To alleviate the above problem, we observe that by processing nearby solid samples using consecutive threads, it may be possible to improve cache hit rates for fluid data accesses and improve memory performance. Based on this idea, we define a parameter ℓ and partition the fluid grid around the solid into *blocks* of size $\ell \times \ell \times \ell$, ordering these blocks by the natural $x-y-z$ order. We then order the solid samples based on the block they lie in, so that samples from an earlier block are stored before samples from a later block. Within each block, we order the samples using Z-ordering, by computing the Morton code [102] for each sample and sorting the codes within the block (cf. Fig. 7c). We determine the optimal ℓ using our parametric cost model, as described later in Section 4.5. Note that this is a new data layout for improving coalescing, and has not been proposed in previous IB-based implementations of the kinetic method.

To explain the utility of our proposed sample ordering for improving memory performance, we refer to Fig. 8 for a 2D illustration. Assume in this example that a GPU warp contains 8 threads. In Fig. 8a we use $\ell = 1$ whereas in Fig. 8b we use $\ell = 2$. The black points in both subfigures

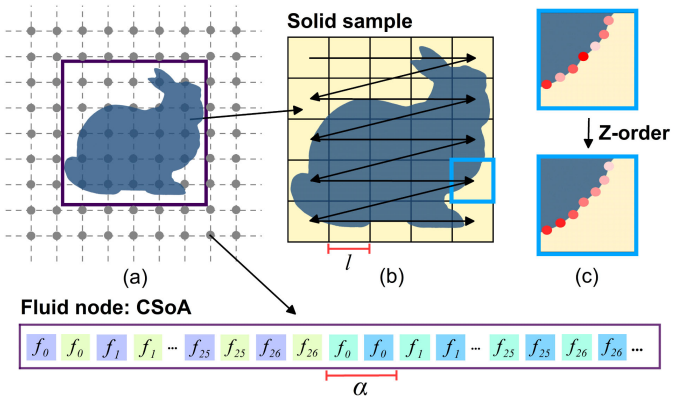


Fig. 7. *2D illustration of the data layout for fluid nodes and solid samples in memory.* To store solid samples, we (a) first find the bounding box of the solid, and then (b) divide the box into blocks of size $\ell \times \ell$ (in 3D we use $\ell \times \ell \times \ell$). These blocks are stored in the same order as the fluid nodes, as shown by the zig-zag black arrows. Within each block, solid samples are first stored in the order they are sampled based on the original triangulation. To further improve memory access performance, these solid samples are Morton-coded and stored in Z-order, where the shading of each sample in (c) represents the ordering of the samples in memory (lighter colors indicate smaller indices). The fluid nodes are stored in the CSoA format, with interpolation parameter α specifying the number of repeating f_i's, i.e. the number of fluid nodes in each group.

show the solid samples processed by one warp, with one thread assigned to each sample. Note that due to the larger ℓ, samples are more localized in Fig. 8b. Now, assume that 3 fluid nodes are stored in each segment of GPU memory. Fig. 8a shows that 4 memory segments, marked by the orange rectangles, need to be read to load the fluid data needed by the warp of solid samples when $\ell = 1$, whereas in Fig. 8b only 3 memory segments need to be read for $\ell = 2$. This shows that differences in the block size can cause variations, sometimes quite significant, in the amount of memory bandwidth consumed and cache hits obtained.

4.3 Reducing Load Imbalance and Warp Divergence

As stated earlier, the use of the IB method removes the conditional branching which occurs during boundary treatment in the bounce-back schemes used in many kinetic solvers. However, a baseline implementation of the IB method requires different numbers of loop iterations for

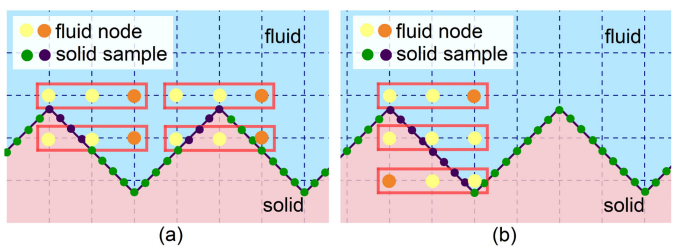


Fig. 8. *2D illustration of caching for different block sizes.* Assume each warp has 8 threads. In both (a) and (b), the purple points represent solid samples processed by threads in one warp, with one thread assigned to each point. When $\ell = 1$ as in (a), the points are dispersed, whereas for $\ell = 2$ as in (b) the points are more localized. Now, assume 3 fluid nodes are stored in each memory segment. In (a) and (b), each rectangle marks the fluid nodes stored in one memory segment. In (a), four memory segments need to be read to load the fluid data needed by the solid samples, while in (b) only three segments are needed due to greater sample locality. This shows how varying ℓ can improve memory bandwidth and caching performance.

TABLE 1
Comparison of the Number of Loop Iterations Using "Gathering" and "Scattering", and the Improvement in Performance

	number of loops		time (ms)
	min.	max.	
gathering with grid acceleration	2	754	53.9
scattering with atomic addition	8	27	6.1

different fluid nodes around the solid boundary, with the variation often being quite large, and this leads to substantial load imbalance between the threads. In this section, we propose additional changes to the traditional IB implementation to improve load balancing.

4.3.1 Load Imbalance in the Immersed Boundary Method

We first show how a baseline implementation of the IB method can result in load imbalance among threads. Consider the force spreading step described in Section 3.1 (cf. Eq. (8)). Here, threads are assigned to fluid nodes and the penalty force at a fluid node is computed by summing together all the penalty forces, computed immediately after interpolation by Eq. (6), at nearby solid samples (this step is also called "gathering" [103]), weighted by a kernel K in a $2 \times 2 \times 2$ neighborhood in 3D (2×2 in 2D), as shown in the blue box in Fig. 4 for the 2D case. While Poisson-disk sampling ensures almost uniform samples over the solid surface, each $2 \times 2 \times 2$ neighborhood may contain local surfaces with different areas due to variations in local geometry, resulting in large variance in the number of samples in the force summation at different fluid nodes (see Table 1 for the variation in the number of loop iterations in an example computation). This problem is especially severe when the solid geometry is complex and local geometric roughness changes with location.

4.3.2 Load Balancing the Immersed Boundary Method

While we saw above that parallelizing across fluid nodes can result in substantial load imbalance, the imbalance can be significantly reduced if we instead parallelize over the solid samples. This technique is also called "scattering", and has been used in other GPU-based particle simulations, e.g. [103]. In particular, we assign one GPU thread to process each solid sample. In the interpolation process, each sample interpolates fluid velocities from nearby grid nodes, while in the force spreading process, each solid sample redistributes its penalty force to nearby fluid nodes, as shown in the red box in Fig. 4. Eq. (8) is equivalent to adding a penalty force with the corresponding kernel weight $K(\|\boldsymbol{x}_s - \boldsymbol{x}_f^i\|)\boldsymbol{g}^{s \to f}(\boldsymbol{x}_s)$ to the fluid velocities of all grid nodes $\boldsymbol{u}(\boldsymbol{x}_f^i)$ in the same $2 \times 2 \times 2$ (or 2×2) neighborhood. While the number of fluid nodes still varies for each solid sample, the variation is much less than the variation in the number of solid samples in each neighborhood, leading to greatly reduced load imbalance (see Table 1 for the reduction in load imbalance using scattering). Note that since multiple

solid samples may try to add forces to the same fluid node, we use atomic additions to avoid race conditions.

4.3.3 Domain Boundary Treatment

The method described above significantly reduces load imbalance for the IB method at solid boundaries. However, special treatment is still needed for the domain boundary. Since the IB method is difficult to apply at domain boundaries, at these locations we resort to the traditional bounce-back scheme. Recall that the main drawback of this method is that it causes warp divergence due to conditional branching. To mitigate this effect, we perform bounce-back using 6 different kernels for the 3D case (4 different kernels for 2D case), one for each face of the domain boundary. For each face, threads only need to check one condition, namely whether they border the face being processed, while the distribution functions which need bounce-back treatment are known in advance and do not need to be determined using conditionals. Overall, performance is improved using multiple kernel launches instead of a single one.

4.4 Increasing Occupancy

GPU occupancy refers to the number of active GPU threads, which should be maximized in order to make full use of the processing capabilities of the GPU. One factor which limits occupancy is the amount of resources, such as registers or shared memory, needed by each thread. The ACM-MRT-based kinetic solver we employ produces significantly higher quality results compared to other kinetic methods. However, this comes at the cost of needing to perform very lengthy arithmetic computations in the collision process, which in turn requires a large number of GPU registers and results in low occupancy. Thus, to improve performance, it is important that we relieve register pressure.

4.4.1 Using Multiple Kernel Launches

One straightforward strategy for reducing the number of required registers is to separate the most complex part of the collision computation, namely the calculation of the Ω_i ($i \in \{0, \ldots, 26\}$) values, into multiple kernel launches. That is, each kernel launch only computes a portion of the Ω_i values. As we increase the number of kernel launches, the registers needed for each thread decreases. However, each kernel launch requires reading and writing partially computed f_i values from and to the GPU main memory. Thus, a tradeoff exists between the higher occupancy enabled by a larger number of kernel launches and the cost of a larger number of memory accesses. In practice, we found that using two kernel launches typically achieved the best balance and highest performance in our GPU system.

4.4.2 Simplification of Relaxation Computation

In addition to separating the computation of different Ω_i's into multiple kernels, another strategy to reduce the number of required registers is to simplify the lengthy arithmetic expressions used in computing collision. The specific form of the expression for one of the collision terms used in the ACM-MRT model is given in the *supplementary document*, available online, and will not be covered in detail here.

TABLE 2
Example of Monomials and Their Corresponding Weights Used in Computing M^{-1} in the ACM-MRT Collision Model

	$\boldsymbol{u}_y$	...	$\boldsymbol{u}_x\boldsymbol{u}_y{}^2$	...	$\boldsymbol{u}_x{}^2\boldsymbol{u}_y{}^2\boldsymbol{u}_z{}^2$
Ω_0	$2(\tilde{f}_{12} - \tilde{f}_{25})$	...	$-\tilde{f}_{10} + \tilde{f}_{13}$	...	$-\rho$
Ω_1	$-\tilde{f}_5 - ... + \tilde{f}_{25}$	...	$-0.5\rho + ... - 0.5\tilde{f}_{13}$	...	0.5ρ
...	...	...	...	...	...
Ω_{13}	$0.5\tilde{f}_5 - ... - 0.5\tilde{f}_{25}$	...	$0.5\tilde{f}_5 + ... + 0.25\tilde{f}_{13}$	...	-0.25ρ
Ω_{14}	$-0.5\tilde{f}_5 + ... - 0.5\tilde{f}_{25}$	...	$0.5\tilde{f}_5 + ... + 0.25\tilde{f}_{13}$	...	-0.25ρ
...	...	...	...	...	...

However, a key step involves the computation of M^{-1}, an example of which is given by the expression shown below. Note that the following formula only shows about one sixth of the complete expression for Ω_{26}.

$$
\begin{aligned}
\Omega_{26} =& \tilde{f}_{20}/8 - \tilde{f}_{16}/8 + \tilde{f}_{21}/8 + \tilde{f}_{22}/8 + \cdots \\
&+ (\tilde{f}_{24}\boldsymbol{u}_y)/4 - (\tilde{f}_4\boldsymbol{u}_z)/8 + (\tilde{f}_{10}\boldsymbol{u}_z)/16 \cdots \\
&+ (\tilde{f}_{25}\boldsymbol{u}_z)/4 + (\tilde{f}_6\boldsymbol{u}_x{}^2)/8 - (\tilde{f}_{11}\boldsymbol{u}_x{}^2)/16 \cdots \\
&- (\tilde{f}_{19}\boldsymbol{u}_y{}^2)/16 + (\tilde{f}_4\boldsymbol{u}_z{}^2)/8 - (\tilde{f}_{10}\boldsymbol{u}_z{}^2)/16 \cdots \\
&- (\tilde{f}_7\boldsymbol{u}_x{}^2\boldsymbol{u}_z{}^2)/12 + (\tilde{f}_8\boldsymbol{u}_x{}^2\boldsymbol{u}_z{}^2)/24 \cdots \\
&- (\tilde{f}_{16}\boldsymbol{u}_x\boldsymbol{u}_y)/2 + (\tilde{f}_{22}\boldsymbol{u}_x\boldsymbol{u}_y)/2 + (\tilde{f}_4\boldsymbol{u}_x\boldsymbol{u}_z)/4 \cdots \\
&+ (\tilde{f}_{21}\boldsymbol{u}_x\boldsymbol{u}_z)/2 + (\tilde{f}_4\boldsymbol{u}_y\boldsymbol{u}_z)/4 + (\tilde{f}_5\boldsymbol{u}_y\boldsymbol{u}_z)/4 \cdots \\
&- (\tilde{f}_5\boldsymbol{u}_x\boldsymbol{u}_y{}^2)/4 - (\tilde{f}_6\boldsymbol{u}_x{}^2\boldsymbol{u}_y)/4 - (\tilde{f}_7\boldsymbol{u}_x\boldsymbol{u}_y{}^2)/24 \\
&- (\tilde{f}_7\boldsymbol{u}_x{}^2\boldsymbol{u}_y)/24 + (\tilde{f}_8\boldsymbol{u}_x\boldsymbol{u}_y{}^2)/12 + (\tilde{f}_8\boldsymbol{u}_x{}^2\boldsymbol{u}_y)/12 \cdots \\
&- (\tilde{f}_4\boldsymbol{u}_x\boldsymbol{u}_z{}^2)/4 - (\tilde{f}_6\boldsymbol{u}_x{}^2\boldsymbol{u}_z)/4 + (\tilde{f}_7\boldsymbol{u}_x\boldsymbol{u}_z{}^2)/12 \\
&+ (\tilde{f}_7\boldsymbol{u}_x{}^2\boldsymbol{u}_z)/12 - (\tilde{f}_8\boldsymbol{u}_x\boldsymbol{u}_z{}^2)/24 - (\tilde{f}_8\boldsymbol{u}_x{}^2\boldsymbol{u}_z)/24 \cdots \\
&- (\tilde{f}_4\boldsymbol{u}_y\boldsymbol{u}_z{}^2)/4 - (\tilde{f}_5\boldsymbol{u}_y{}^2\boldsymbol{u}_z)/4 - (\tilde{f}_7\boldsymbol{u}_y\boldsymbol{u}_z{}^2)/24 + \cdots \\
&- (\tilde{f}_7\boldsymbol{u}_y{}^2\boldsymbol{u}_z)/24 - (\tilde{f}_8\boldsymbol{u}_y\boldsymbol{u}_z{}^2)/24 - (\tilde{f}_8\boldsymbol{u}_y{}^2\boldsymbol{u}_z)/24 \cdots
\end{aligned}
\tag{10}
$$

Here, $\tilde{f}_i = [\boldsymbol{D}\boldsymbol{M}(\boldsymbol{f} - \boldsymbol{f}^{eq})]_i$, and $[\cdot]_i$ picks the i'th element out of a vector. Computing this and similar expressions requires dozens of registers for each GPU thread. This can be reduced with the observation that the expressions for different Ω_i's share many similar terms, with the only difference being the coefficients. In general, all the terms appearing in an expression can be written as the sum of monomials of the form $\omega(\rho)\prod_{\alpha=0}^{2}\boldsymbol{u}_\alpha^{p_\alpha}$ ($p_\alpha = 0, 1, 2$ is the order of the α'th component). Different Ω_i's may have different combinations of orders with different weights $\omega(\rho)$. Within each Ω_i, any term having the same order of p_0, p_1, p_2 but different weights $\omega(\rho)$ can be merged, as shown in Table 2. Across different Ω_i's, we first compute the values of the shared terms and store them. Later, we simply read back the values needed for each Ω_i.

4.4.3 Discussion

Based on our experiments for the test case shown in Fig. 9, using a grid resolution of $100\times200\times100$ with 45,127 solid samples, collision performance is improved by around 35

Fig. 9. An example of a jet flow passing over a slanted airplane obstacle was used in the performance comparisons in Figs. 11, 12, 13, 14, and 15.

percent using a combination of multiple kernel launches and simplified collision terms, while arithmetic expression simplification alone improves performance by 6 percent.

4.5 Our Parametric Model

In the preceding subsections, we described a number of different GPU optimizations for improving the performance of our kinetic solver. While the optimizations may appear to be largely independent, they are in fact interconnected. In particular, the solver's performance is strongly affected by the memory layout, which in turn is determined by the parameters ℓ, giving the size of blocks that solid samples are grouped into, and α, giving the granularity of the CSoA layout for fluids. We therefore combine these parameters into a metric $H^t(\ell, \alpha)$ measuring the cost to simulate a time step starting from time t. Note that $H^t(\ell, \alpha)$ may vary for different t, as the immersed solid may lie at different orientations over time, leading to different costs to implement the IB method. To obtain more stable measurements, we consider measuring the average cost over a time interval $[t_s, t_e]$. We frame this as searching for optimal values for ℓ and α such that the following cost is minimized:

$$\text{argmin}_{\ell,\alpha} \frac{1}{N_{t_s \to t_e}} \sum_{t=t_s}^{t_e} H^t(\ell, \alpha), \tag{11}$$

where $N_{t_s \to t_e}$ is the number of time steps between t_s and t_e. Note that we may use different numbers of time steps in order to obtain optimal parameters balancing different cases in a simulation. For static solids, $N_{t_s \to t_e}$ is typically small (e.g., $N_{t_s \to t_e} = 10$), while for dynamic solids, $N_{t_s \to t_e}$ should be large enough to cover many different orientations of the solid. Since translation of solid usually does not strongly affect the objective, we consider $N_{t_s \to t_e}$ only when rotating the solid object by an angle of 2π in 3D, leading to a value of around 500 for $N_{t_s \to t_e}$. This minimization is performed once before the start of every simulation with a different setting for the grid resolution, geometry shape, etc.

To minimize Eq. (11), we enumerate all possible combinations of ℓ and α and search for the combinations with minimal cost. Since ℓ is the side length of the subdivided block, it is upper bounded by the smallest edge length L_m of the solid's bounding box, and thus we search for $\ell \in \{1, 2, \ldots, L_m\}$. For the CSoA parameter α, we only consider powers of 2 since we want α to be divisible by or divisible into the thread warp size of 32. We thus search for $\alpha \in \{2^1, 2^2, \ldots, 2^{\lfloor \log_2 N_f \rfloor}\}$, where N_f is the total number of fluid nodes.

Fig. 10 shows the simulation cost in seconds for certain values of ℓ and α and for different types of geometries. We note that after minimizing Eq. (11), we can obtain the best parameters for the scenario we want to simulate. But if the scenario changes, e.g., if we use a different size for the fluid domain, a different solid geometry, or if the solid is sampled with a different number of points, we should perform the minimization again. The cost of the procedure is usually not excessive, e.g., 5 to 15 minutes at a grid resolution of $200 \times 400 \times 200$, compared to a total simulation time of several hours. As the simulator can sometimes be several times faster using a good parameter setting compared to a poor one, it is usually well worthwhile to perform optimal parameter search before starting a simulation.

Algorithm 2. Our GPU-Optimized IB-Based Kinetic Solver

```
// Find optimal parameters for ℓ and α.
SearchOptimalParameters(ℓ, α);                          ▷ Section 4.5
ReorganizeMemory(ℓ, α);          ▷ Eq. (9) for arrays of vectors
while time step ≤ maximum time step do
  // (a): Perform parallel streaming.                   ▷ Section 3.2
  parallel for each fluid node x_f do
    for each i ∈ [0, 26] of node x_f do
      j = RemapIndex(x_f, i);                           ▷ Eq. (9)
      j_n = RemapIndex(x_f − c_i, i);                   ▷ Eq. (9)
      f^t[j] = f^{t−1}[j_n];                            ▷ Eq. (3)
    end for
  end for
  // (b): Compute macroscopic variables in parallel.   ▷ Section 3.2
  parallel for each fluid node x_f do
    UpdateDensityAndVelocity(x_f);                      ▷ Eq. (2)
    ResetForce: g[x_f] = 0;
  end for
  // (c): Immersed boundary treatment.                 ▷ Section 4.3
  parallel for each solid sample x_s do
    ComputePenaltyForce(x_s);                           ▷ Eq. (7)
  end for
  parallel for each solid sample x_s do
    SpreadPenaltyForce(x_s);      ▷ Eq. (8) (atomic addition)
  end for
  ParallelTotalForceAndTorqueCalculation();
  ParallelSolidSampleUpdates();
  // (d): Perform parallel collision.                  ▷ Section 4.4
  parallel for each fluid node x_f do
    Compute m[x_f] and m^{eq}[x_f];
  end for
  parallel for each fluid node x_f do
    Compute Ω_{0−13}[x_f] and add G_{0−13}[x_f];        ▷ Eq. (5)
  end for
  parallel for each fluid node x_f do
    Compute Ω_{14−26}[x_f] and add G_{14−26}[x_f];      ▷ Eq. (5)
  end for
  // (e): Process domain boundary in parallel.         ▷ Section 4.3.3
  ParallelLeftBoundaryTreatment();
  ParallelRightBoundaryTreatment();
  ParallelUpBoundaryTreatment();
  ParallelDownBoundaryTreatment();
  ParallelForwardBoundaryTreatment();
  ParallelBackBoundaryTreatment();
end while
```

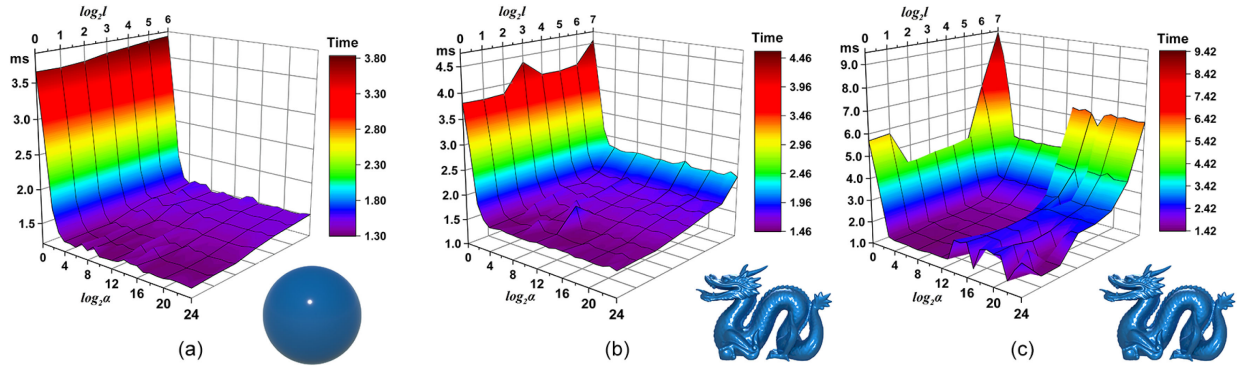


Fig. 10. *Performance variations from changing model parameters for different solid geometries.* Given a simple geometry such as the ball in (a), the simulation cost does not change much as we vary the sample block size ℓ and the CSoA parameter α within a specific range ($\ell \in [2, 4]$, $\alpha \geq 64$). However, for more complex geometries such as the dragon, the cost can vary significantly based on (b) the degree of concavity in the geometry compared to (a) and (c) the number of solid samples compared with (b) ((c) contains around 10 times more solid samples than (b)). This indicates that searching for the optimal ℓ and α is important for complex geometries containing a large number of solid samples.

4.6 Performance Analysis

We summarize our entire optimized kinetic solver in Algorithm 2. In the rest of this section, we analyze the performance of different parts of the solver. We compared the performance of our optimized solver with the baseline solver described in Section 3.2, which is for the most part a faithful implementation of the algorithm described in [1], except that [1] did not use the immersed boundary method. To make our analysis as detailed as possible, we decomposed our solver into its main subroutines (cf. Section 3.2), consisting of streaming, macroscopic fields calculation, the IB method, collision, and domain boundary treatment. We analyzed each of these components using key GPU performance metrics including thread occupancy, memory load/store efficiency, warp divergence and warp execution efficiency.

Figs. 11, 12, 13, 14, and 15 show performance comparisons of different parts of our optimized solver and the baseline solver using the scene setup shown in Fig. 9. We see that our parameterized data layout significantly improves memory access efficiency. Occupancy was also improved several fold during the IB and collision computations, while warp execution efficiency was improved by a factor of three during the IB computation. The overall performance was improved by a factor of 5 to 10 compared to the straightforward baseline implementation.

4.7 Extension to Multi-GPU Systems

The optimizations described in the previous sections were performed on a single GPU, limiting the resolution at which the kinetic solver can simulate. To support high resolution scenarios for large-scale simulations, we modify the algorithm to run in parallel on multiple GPUs. Since the kinetic solver uses only local information from one-hop neighbors, we can also easily extend the optimizations techniques described earlier to multi-GPU systems.

We consider a setting where m GPUs (in our case $m = 4$) are installed on a single cluster node (the multi-node case is more complex and we leave it for future work). We first subdivide the fluid nodes in the simulation domain into multiple regions along one dimension (e.g., the z-dimension), as shown in Fig. 16, while ensuring that each region contains roughly the same number of fluid nodes. For each pair of neighboring regions, we extend both regions by a one-cell-wide "ghost region" (marked in red in Fig. 16), holding data from the shared face (marked in blue) with its neighbors. Data is updated (copied) between neighboring GPUs after

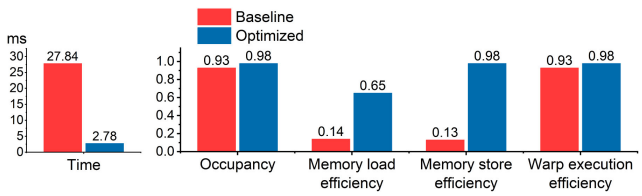


Fig. 11. Comparison of streaming performance with (blue bar) and without (red bar) our optimizations.

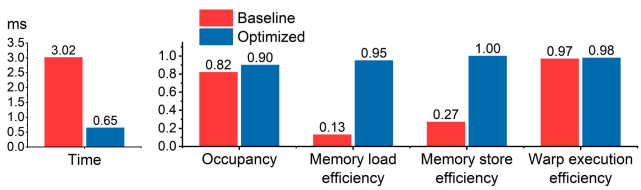


Fig. 12. Comparison of performance for calculating macroscopic variables with (blue bar) and without (red bar) our optimizations.

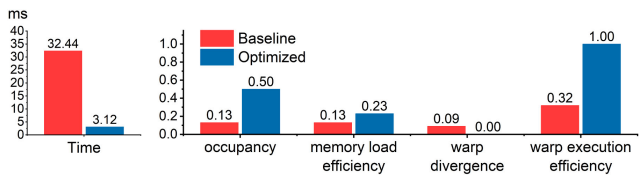


Fig. 13. Comparison of immersed boundary computation performance with (blue bar) and without (red bar) our optimizations.

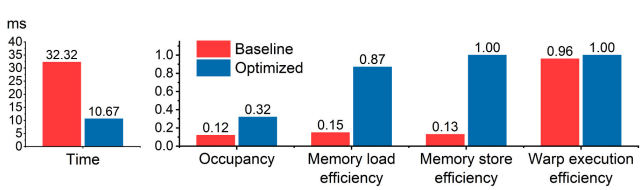


Fig. 14. Comparison of collision performance with (blue bar) and without (red bar) our optimizations.

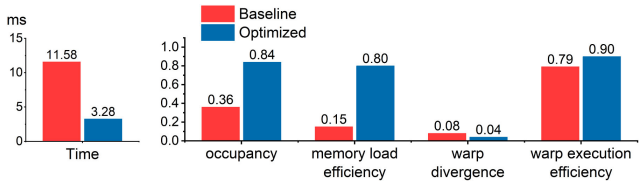


Fig. 15. Comparison of domain boundary performance with (blue bar) and without (red bar) our optimizations.

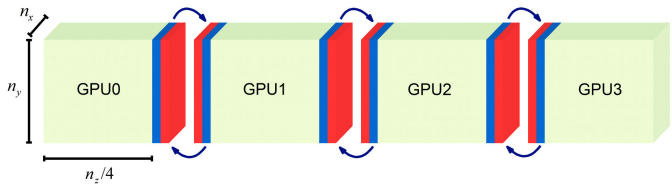


Fig. 16. *Multi-GPU simulation.* The whole fluid domain is subdivided into m ($m \leq 4$ in our case) regions of roughly equal size along the z-dimension. We extend the shared face between neighboring GPUs by one cell along the z-direction and copy data from the neighboring GPUs (the blue regions), with arrows indicating the direction of copying. n_x, n_y and n_z are numbers of fluid grid nodes in the x-, y- and z-directions.

every simulation time step. This communication can be done using GPU to GPU memory transfer when supported by hardware. However, due to limitations of our current hardware and for the sake of implementation portability, our current implementation performs data transfer through the CPU. Note that this may substantially reduce performance, especially for large m, and that with faster inter-GPU communication mechanisms all the results described below are expected to improve. The optimizations described earlier can be performed independently within each region. When immersed solid does not move, we can subdivide solid samples into m groups based on regions they lie in. However, when the solid is allowed to move, its sample points may reside in different regions over time. To deal with this situation, we store all solid samples in every GPU, thus avoiding the need to copy solid samples among GPUs, while incurring the cost of a slight increase in memory usage.

5 Results and Discussions

We implemented the single GPU version of our solver on an NVIDIA TITAN XP GPU with 12 GB of memory, installed on an 8-core Intel Xeon E5-2650 workstation with 64 GB of memory. For the multi-GPU version of our algorithm, we used a system with 4 NVIDIA Tesla P40 GPUs per node, each having 24 GB of memory. The system also had 4 Intel Xeon E5-2620 CPUs (8 cores per CPU) with 384 GB of memory. Optimizing model parameters, as described in Section 4.5, usually took 2 to 5 minutes for flows with static solids and 5 to 15 minutes for flows with dynamic solids at a grid resolution of $200 \times 400 \times 200$. To render the simulated flows, we usually injected smoke particles, which were traced in parallel on the GPU. We then projected the particle densities into a uniform volume grid, and used Mitsuba [104] to perform high quality rendering. The specific parameter settings for the simulations shown in this paper and the related performance statistics are reported in Table 3.

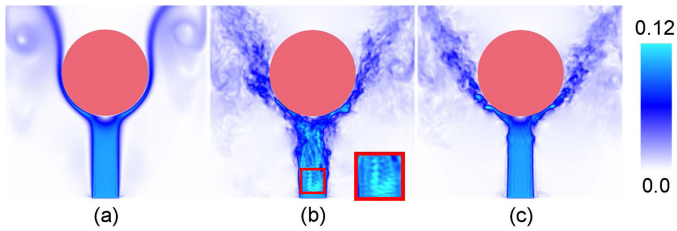


Fig. 17. *Accuracy comparison for kinetic solvers using different collision models.* Visualization of velocity magnitude from a cross-section of a 3D velocity field for a jet flow impinging on a ball obstacle: (a) using the BGK model with the smallest stable viscosity (2×10^{-3}), (b) using the MRT model with the smallest stable viscosity (10^{-4}), and (c) using the ACM-MRT model with the same viscosity as (b). The color bar shows the ranges of velocity magnitudes in LBE scale.

5.1 Quality Comparison of GPU-Based Kinetic Solvers

Recent GPU optimized kinetic solvers have typically employed the BGK or RM-MRT [105] collision models. In this section, we compare the quality of simulations obtained using these models with that using the ACM-MRT model. We also compare the performance of these models.

To ensure fairness in the quality comparisons, we use the D3Q27 lattice for all three collision models. Fig. 17 shows velocity fields obtained from the D3Q27 BGK, RM-MRT and ACM-MRT collision models. Since the BGK model is not stable at high Reynolds numbers, we set its Reynolds number as high as possible while remaining stable. Despite this, we still obtain an unnaturally smooth result, as shown in Fig. 17a. The RM-MRT model can be stable at high Reynolds numbers but will be very dispersive, as seen in Fig. 17b, and exhibit unnatural oscillation artifacts, as shown in the magnified inset. Finally, the ACM-MRT model can support very high Reynolds numbers with very limited diffusion and dispersion, and produces the most visually appealing result, as shown in Fig. 17c.

When comparing performance, we first note that to our knowledge, there are no GPU-optimized kinetic solvers using the D3Q27 lattice structure, and also no GPU-based solvers using the immersed boundary method. For these

TABLE 3
Parameter Settings and Performance Statistics for Different Simulations Shown in the Paper

Figures.	Grid resolution	No. of solid samples	ℓ	$\log_2\alpha$	No. of GPUs	Storage	time / Δt	total time steps	total time cost
Fig. 1	1200×250×840	3,688,157	2	28	4	58.4 GB	1.21 sec.	76,600	1544.8 min.
Fig. 23a	400×200×400	48,336	2	9	1	7.4 GB	0.56 sec.	20,000	186.7 min.
Fig. 23b	400×200×400	53,104	4	16	1	7.4 GB	0.58 sec.	25,600	247.5 min.
Fig. 24	2240×320×480	1,971,941	2	29	4	79.5 GB	1.70 sec.	26,300	745.2 min.
Fig. 25	2100×240×600	909,788	2	28	4	69.9 GB	1.43 sec.	41,500	989.1 min.

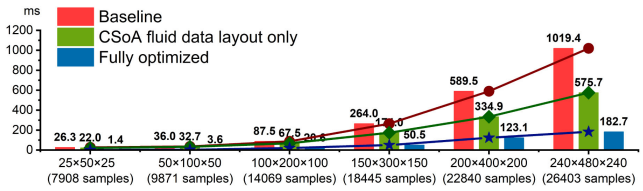


Fig. 18. *Single GPU performance comparisons.* Red bars: the baseline kinetic solver (cf. Section 3.2). Green bars: the modified baseline kinetic solver using CSoA data layout for fluid nodes. Blue bars: our GPU-optimized kinetic solver. The comparisons were done with simulations at different grid resolutions and with resolution-matched numbers of solid samples in order to maintain simulation accuracy.

reasons, it is difficult to find suitable existing works against which to compare the performance of our optimized solver. Instead, we have chosen to implement D3Q27 versions of the BGK and RM-MRT models using an optimized CSoA data layout to compare with our solver. We now give a detailed analysis showing the performance advantages of our solver in both single and multi-GPU settings for various resolutions of the simulation domain.

5.1.1 Single GPU Comparison

We first compare the performance of our solver on a single GPU with the baseline solver described in Section 3.2, and a solver using an optimized CSoA data layout for fluid nodes, but employing none of the other optimizations described in Section 4. Fig. 18 shows the performance of the three solvers at different grid resolutions. The baseline implementation suffers from uncoalesced memory accesses, and its cost increases significantly as resolution increases. The CSoA-based solver is able to coalesce memory accesses for fluid nodes but does not optimize the IB method, so it also shows limited performance as the resolution increases. Finally, our optimized solver shows increasing performance improvements relative to the other two solvers as resolution increases, and is over $3.1\times$ and $5.5\times$ faster than the CSoA and baseline solvers at the highest resolution we tested.

We also present another comparison of the BGK, RM-MRT and ACM-MRT implementations in which we employed the same optimizations described in Section 4 for all three models, and only vary how the collision operation is performed. The results are shown in Fig. 19. The plot shows that our optimized ACM-MRT implementation has nearly the same computational efficiency as the significantly simpler BGK model, while producing much more visually appealing outputs. In general, the cost of arithmetic calculations occupies a relatively small portion of the overall running time, indicating

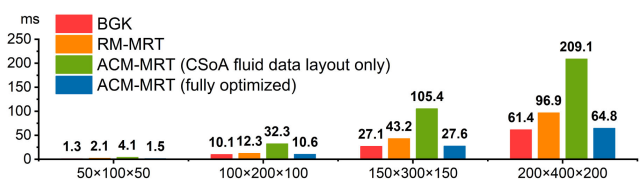


Fig. 19. *Performance comparisons for different collision models.* We show timings for the collision step only, as the other parts of the different models employ the same optimizations and thus have nearly the same performance. At different resolutions, our GPU-optimized ACM-MRT model performs nearly as efficiently as the basic BGK model, but produces much more accurate results for turbulent flows.

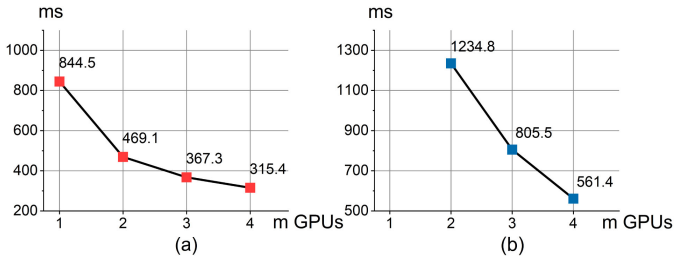


Fig. 20. *Performance comparison for multi-GPU simulations.* In (a), we use a grid resolution of $200 \times 400 \times 200$ and 1,579,468 solid samples on 1 to 4 GPUs. In (b), we increase the simulation size to $400 \times 400 \times 400$ with the same number of solid samples, and run on 2 to 4 GPUs. Data copying has a larger impact on the smaller simulation, while the higher-resolution simulation shows improved scalability.

that efficient memory accesses play a key role in optimizing kinetic solvers on the GPU.

5.1.2 Multi-GPU Comparison

In this section, we analyze the performance of our optimized kinetic solver running on different numbers of GPUs at the same grid resolution. As stated earlier, communication among GPUs is currently done by copying data through the CPU, and we expect that direct GPU to GPU communication can lead to substantial performance improvements.

Fig. 20a shows the change in runtime when we perform a simulation of size $200 \times 400 \times 200$ using 1 to 4 GPUs. We observed nearly perfect scaling when going from 1 to 2 GPUs. However, as the number of GPUs increases, the communication cost and serialization at the CPU start to dominate, leading to reduced scaling. To better understand the impact of communication on performance, we also performed a simulation of size $400 \times 400 \times 400$ on 2 to 4 GPUs as shown in Fig. 20b. Since we subdivided the simulation domain along the z-axis, the amount of communication between neighboring GPUs doubled in the new simulation while the total amount of computation quadrupled. Thus, communication is expected to have smaller impact on overall performance. This is indeed the observed behavior, as scaling from 2 to 3 and 4 GPUs leads to improvements of $1.5\times$ and $2.2\times$ resp. in the larger-scale simulation, and only $1.27\times$ and $1.49\times$ in the smaller-scale experiment. Thus, we expect that the scaling behavior can be further improved as we perform simulations at even larger scales.

5.2 Comparison With GPU-Based INSE Solvers

To further highlight the advantages in performance and accuracy of our GPU-optimized kinetic solver, we compared it with the recently proposed reflection-advection MacCormack (MC+R) solver [7], which we implemented on the GPU using an SoA data layout and using NVIDIA's optimized cuSPARSE library [74] for solving sparse linear systems. We note that a newer method called "BiMocq^2" [8] can better preserve turbulence details and possibly offer enhanced performance compared to MC+R on the GPU. However, there is no publicly available full GPU implementation of the BiMocq^2 method, making it difficult to compare against. In addition, based on the results from [8] at similar grid resolutions, the method does not produce higher visual quality compared to our kinetic method. Thus, in the

Fig. 21. *Visual comparison of fluid solvers at different settings.* MC+R INSE solver with time step sizes (a) $40\times$, (b) $10\times$ and (c) equal to our solver. (d) Output of our kinetic solver using the ACM-MRT model. (e) Output of our kinetic solver using a coarser grid resolution (half the resolution in each dimension). For the simulations in (a) to (d), the grid resolution is $200\times400\times200$. All these simulations are implemented on GPUs using our optimizations. Note that (c) and (d) are visually similar (however, our kinetic solver can produce even finer details), while the INSE solver with larger time step sizes have much different smoke patterns due to temporal smoothing, smearing out turbulence details. The simulation in (a) has similar speed as ours, while (b) and (c) perform $5\times$ and $50\times$ slower respectively than our solver. In addition, due to its higher convergence rate, our kinetic solver operating at a lower grid resolution in (e) still produces similar smoke patterns to those shown in (d), but runs 6 times faster than (d) and 300 times faster than (c). These results demonstrate the strength of our kinetic solver in both performance and visual quality.

following performance analysis we base our comparison against the GPU-based MC+R INSE solver.

To ensure accuracy, especially for turbulent flow simulations, the preconditioned conjugate gradient (PCG) algorithm [106] was used to solve the sparse linear systems using a sufficient number of iterations. At a resolution of $200\times400\times200$, we found that at least 300 iterations were needed for proper simulation of turbulent flows when balancing accuracy and efficiency. The performance of INSE solvers can be improved by using larger time step sizes while still producing visually plausible results, especially in scenarios such as real time applications. However, we found that in turbulent flow simulations, larger time step sizes sacrifice temporal accuracy and lead to results which deviate substantially from the reference output. Below, we compare our simulation with those from the MC+R INSE solver at different time step sizes.

Fig. 21 shows a visual comparison between our kinetic solver and an MC+R INSE solver using different time step sizes. Clearly, using a time step size which is (a) 40 times (Fig. 21a) or (b) 10 times (Fig. 21b) larger than that used in our kinetic solver (Fig. 21d) can improve the INSE solver's performance to a degree that it becomes comparable to ours. However, these step sizes also produce flow patterns which deviate substantially from the reference and lose large amounts of turbulence details. The INSE simulation in Fig. 21c uses the same time step size as ours and produces similar flow patterns, which indicates the accuracy of our solver. However, our solver has far superior performance, as demonstrated in Fig. 22 for different grid resolutions and with different time step sizes.

We note that the performance of INSE solvers can be further improved by techniques such as multi-grid methods. Based on an existing report [78], multi-grid methods can typically accelerate INSE solvers by 10 to 20 times on a multi-core CPU, and can be expected to run even faster on a GPU. However, a multi-grid INSE solver is still less scalable over multiple GPUs than our kinetic solver and thus less efficient when grid resolution is very high. In addition, our solver converges faster than many existing INSE solvers used in the graphics domain (as shown in e.g. [2]), implying that we can use lower grid resolutions while still producing visually similar results. Fig. 22e shows a simulation with our solver using only half the resolution in each dimension (and thus $8\times$ fewer grid nodes) while producing patterns very similar to the higher resolution results in (c) and (d). Note that at the lower grid resolution, our solver is even faster than the MC+R INSE solver using a $40\times$ larger time step size, demonstrating our solver can both run faster and produce higher quality results than state-of-the-art INSE solvers. Lastly, our solver exhibits better scalability than GPU-based INSE solvers, enabling fast simulations at high grid resolutions.

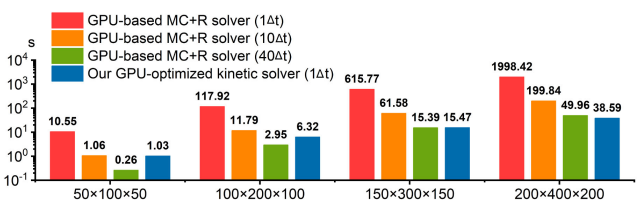


Fig. 22. *Performance comparison with MC+R INSE solver using different time step sizes.* We show the timings (by simulating the same *one physical second*) of a GPU-based MC+R INSE solver with varying time step sizes and our GPU-optimized kinetic solver under different grid resolutions. Due to large variations in timing, we plot the graph using a log-scale. Note the high scalability of our kinetic solver.

5.3 Visual Simulation Results

Using our high-performance kinetic solver, we can efficiently generate a variety of fluid simulation results. Fig. 23 shows two flow simulations of colliding vortices where a ball obstacle is immersed in the center and a rotating torus disturbs the surrounding smoke. To enable high resolution simulations, we ran our solver on a 4-GPU system. Figs. 1, 24, and 25 show high resolution simulations involving complex solid objects (both static and dynamic) computed on the multi-GPU system

Fig. 23. *Smoke simulations at a normal resolution.* (a) simulation of vortices colliding around a ball; (b) simulation of a torus rotating in the air. Both simulations use a grid resolution of $400\times200\times400$.

Fig. 24. *High resolution simulation of airflow around a car.* We employed 4 GPUs to simulate the airflow over a car at a high resolution, displaying realistic wake turbulence details behind the car.

within a relatively short amount of time and displaying detailed and realistic-looking turbulence.

5.4 Model Parameter Variation

The optimal values of model parameters ℓ and α may change depending on the shape of the immersed solid as well as the number of solid samples within each fluid cell. For simple geometries such as a ball or a torus, the cost for different ℓ and α values is similar, as shown in Fig. 10a. The optimal ℓ lies in the range [2,4], while the optimal $\alpha \geq 64 = 2^6$; picking any ℓ and α in this range leads to acceptable performance. However, for complex geometries with high degrees of concavity, the cost surface exhibits significant variations with a number of local peaks, as shown in Fig. 10b. If such a geometry also contains a large number of solid samples, the cost surface becomes even more unpredictable, as shown in Fig. 10c. Thus, conducting an automated search for optimal parameter values before the start of a simulation is very useful in complex simulation scenarios.

5.5 Limitations

While our kinetic solver is in general much faster than INSE solvers and produces equal or superior visual results, one of its drawbacks is the need for larger amounts of memory, typically 3 times more than the MC+R INSE solver at the same grid resolution. In addition, in order to increase GPU occupancy, we employed multiple kernel launches, which further increases the memory requirement. In situations where memory is limited, we can eliminate multiple kernel launches to save memory, but at the cost of lower occupancy and reduced simulation efficiency. However, we note that even in these cases, our solver is still much faster than GPU-based INSE solvers. Furthermore, for high resolution simulations with dynamic solids, we currently store all solid samples in each GPU to avoid communication, which again leads to a moderate increase in memory usage. Finally, our GPU optimizations are currently not suitable for liquid simulations with kinetic solvers. These simulations would need modifications to the kinetic algorithm, such as the use of a volume-of-fluid method, and thus require revising our memory layout. These limitations point toward further research on more versatile and memory-efficient GPU-based kinetic solvers.

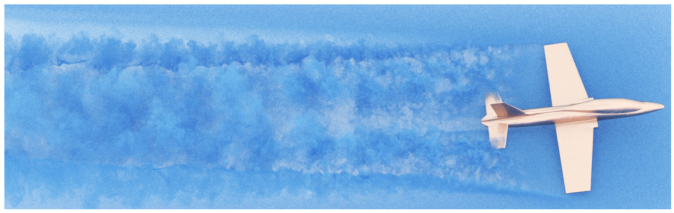


Fig. 25. *High resolution simulation of a moving airplane.* We employed 4 GPUs to simulate an airplane moving through high resolution smoke and generating detailed wake turbulence.

6 Conclusion

In this paper, we proposed GPU optimization techniques to significantly improve the performance of an IB-based kinetic solver using the recently developed ACM-MRT model. Our solver is faster than all other currently available fluid simulation methods, especially for turbulent flow simulations over complex solid geometries. It also produces equal or superior visual quality compared to state-of-the-art INSE solvers. Our optimizations are based on a parametric data layout which considers both fluid nodes and solid samples to improve memory coalescing and bandwidth use. We also proposed a GPU-optimized parallel algorithm for the immersed boundary method to handle boundary conditions, which significantly reduces load imbalance and thread divergence. We used multiple kernel launches with a simplified formulation of the ACM-MRT collision model to improve GPU occupancy. Finally, we combined these optimizations into an integrated cost model to automatically search foroptimal parameter settings to obtain the best performance. We conducted comprehensive comparisons against a range of state-of-the-art solution methods and solvers to validate the high efficiency and visual fidelity of our method. In the future, we will further optimize our solver, including improving its performance in multi-GPU systems and scaling to larger problem sizes.

Acknowledgments

The authors would like to thank all the reviewers for their constructive comments. They also thank Yihui Ma, Chenqi Luo, and Chaoyang Lyu from the FLARE Lab at ShanghaiTech University for helping with video preparations. They are grateful to YOKE Intelligence for providing the 3D reconstruction in Fig. 1 for our large scale simulation. This work was supported by the startup fund of ShanghaiTech University and the National Natural Science Foundation of China under Grant 61976138.

References

[1] W. Li, K. Bai, and X. Liu, "Continuous-scale kinetic fluid simulation," *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 9, pp. 2694–2709, Sep. 2019.
[2] W. Li, Y. Chen, M. Desbrun, C. Zheng, and X. Liu, "Fast and scalable turbulent flow simulation with two-way coupling," *ACM Trans. Graph.*, vol. 39, no. 4, 2020, Art. no. 47.
[3] P. Mullen, K. Crane, D. Pavlov, Y. Tong, and M. Desbrun, "Energy-preserving integrators for fluid animation," *ACM Trans. Graph.*, vol. 28, no. 3, 2009, Art. no. 38.
[4] B. Zhu, W. Lu, M. Cong, B. Kim, and R. Fedkiw, "A new grid structure for domain extension," *ACM Trans. Graph.*, vol. 32, no. 4, 2013, Art. no. 63.
[5] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, "SPH fluids in computer graphics," *Eurographics*, pp. 21–42, 2014.
[6] C. Fu, Q. Guo, T. Gast, C. Jiang, and J. Teran, "A polynomial particle-in-cell method," *ACM Trans. Graph.*, vol. 36, no. 6, 2017, Art. no. 222.

[7] J. Zehnder, R. Narain, and B. Thomaszewski, "An advection-reflection solver for detail-preserving fluid aimulation," *ACM Trans. Graph.*, vol. 37, no. 4, 2018, Art. no. 85.

[8] Z. Qu, X. Zhang, M. Gao, C. Jiang, and B. Chen, "Efficient and conservative fluids using bidirectional mapping," *ACM Trans. Graph.*, vol. 38, no. 4, 2019, Art. no. 128.

[9] X. Shan, X.-F. Yuan, and H. Chen, "Kinetic theory representation of hydrodynamics: A way beyond the Navier-Stokes equation," *J. Fluid Mechanics*, vol. 550, pp. 413–441, 2006.

[10] Y. Guo, X. Liu, and X. Xu, "A unified detail-preserving liquid simulation by two-phase lattice Boltzmann modeling," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 5, pp. 1479–1491, May 2017.

[11] X. Wei *et al.*, "Lattice-based flow field modeling," *IEEE Trans. Vis. Comput. Graphics*, vol. 10, no. 6, pp. 719–729, Nov./Dec. 2004.

[12] N. Thuerey, K. Iglberger, and U. Ruede, "Free surface flows with moving and deforming objects for LBM," *Proc. Vis. Model. Vis.*, vol. 2006, pp. 193–200, Nov. 2006.

[13] N. Thuerey and U. Ruede, "Stable free surface flows with the lattice Boltzmann method on adaptively coarsened grids," *Comput. Vis. Sci.*, vol. 12, no. 5, pp. 247–263, 2009.

[14] S. Chen and G. D. Doolen, "Lattice Boltzmann method for fuid flows," *Annu. Rev. Fluid Mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[15] D. d'Humieres, "Multiple-relaxation-time lattice Boltzmann models in three dimensions," *Philos. Trans. Roy. Soc. London. Series A: Math. Physical Eng. Sci.*, vol. 360, no. 1792, pp. 437–451, 2002.

[16] X. Liu, W.-M. Pang, J. Qin, and C.-W. Fu, "Turbulence simulation by adaptive multi-relaxation lattice Boltzmann modeling," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 2, pp. 289–302, Feb. 2014.

[17] C. S. Peskin, "Flow patterns around heart valves: A numerical method," *J. Comput. Phys.*, vol. 10, no. 2, pp. 252–271, 1972.

[18] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles, "GPU data access on complex geometries for D3Q19 lattice Boltzmann method," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 825–834.

[19] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Optimization of lattice Boltzmann simulations on heterogeneous computers," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 1, pp. 124–139, 2019.

[20] J. Tölke and M. Krafczyk, "Teraflop computing on a desktop PC with GPUs for 3D CFD," *Int. J. Comput. Fluid Dyn.*, vol. 22, no. 7, pp. 443–456, 2008.

[21] S. Chen and G. D. Doolen, "Lattice Boltzmann method for fluid flows," *Annu. Rev. Fluid Mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[22] J. Stam, "Stable fluids," in *Proc. 26th Annu. Conf. Comput. Graph. Interactive Techn.*, 1999, pp. 121–128.

[23] B. Kim, Y. Liu, I. Llamas, and J. Rossignac, "Flowfixer: Using BFECC for fluid simulation," in *Proc. 1st Eurographics Conf. Natural Phenomena*, pp. 51–56, 2005.

[24] R. Wang, H. Feng, and R. J. Spiteri, "Observations on the fifth-order WENO method with non-uniform meshes," *Appl. Math. Comput.*, vol. 196, no. 1, pp. 433–447, 2008.

[25] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, "An unconditionally stable MacCormack method," *J. Sci. Comput.*, vol. 35, no. 2–3, pp. 350–371, 2008.

[26] Q. Cui, P. Sen, and T. Kim, "Scalable Laplacian eigenfluids," *ACM Trans. Graph.*, vol. 37, no. 4, 2018, Art. no. 87.

[27] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke," in *Proc. 28th Annu. Conf. Comput. Graph. Interactive Techn.*, 2001, pp. 15–22.

[28] S. I. Park and M. J. Kim, "Vortex fluid for gaseous phenomena," in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*, 2005, pp. 261–270.

[29] S. Weißmann and U. Pinkall, "Filament-based smoke with vortex shedding and variational reconnection," *ACM Trans. Graph.*, vol. 29, 2010, Art. no. 115.

[30] T. Pfaff, N. Thuerey, J. Cohen, S. Tariq, and M. Gross, "Scalable fluid simulation using anisotropic turbulence particles," *ACM Trans. Graph.*, vol. 29, 2010, Art. no. 174.

[31] T. Brochu, T. Keeler, and R. Bridson, "Linear-time smoke animation with vortex sheet meshes," in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*, 2012, pp. 87–95.

[32] A. Golas, R. Narain, J. Sewall, P. Krajcevski, P. Dubey, and M. Lin, "Large-scale fluid simulation using velocity-vorticity domain decomposition," *ACM Trans. Graph.*, vol. 31, no. 6, 2012, Art. no. 148.

[33] X. Zhang and R. Bridson, "A PPPM fast summation method for fluids and beyond," *ACM Trans. Graph.*, vol. 33, no. 6, 2014, Art. no. 206.

[34] X. Zhang, R. Bridson, and C. Greif, "Restoring the missing vorticity in advection-projection fluid solvers," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 52:1–52:8, Jul. 2015.

[35] R. Bridson, J. Hourihan, and M. Nordenstam, "Curl-noise for procedural fluid flow," *ACM Trans. Graph.*, vol. 26, 2007, Art. no. 46.

[36] T. Kim, N. Thüerey, D. James, and M. Gross, "Wavelet turbulence for fluid simulation," *ACM Trans. Graph.*, vol. 27, 2008, Art. no. 50.

[37] S. Jeong *et al.*, "Data-driven fluid simulations using regression forests," *ACM Trans. Graph.*, vol. 34, no. 6, 2015, Art. no. 199.

[38] M. Chu and N. Thuerey, "Data-driven synthesis of smoke flows with CNN-based feature descriptors," *ACM Trans. Graph.*, vol. 36, no. 4, 2017, Art. no. 69.

[39] Y. Xie, E. Franz, M. Chu, and N. Thuerey, "TempoGAN: A temporally coherent, volumetric GAN for super-resolution fluid flow," *ACM Trans. Graph.*, vol. 37, no. 4, 2018, Art. no. 95.

[40] K. Bai, W. Li, M. Desbrun, and X. Liu, "Dynamic upsampling of smoke through dictionary-based learning," *ACM Trans. Graph.*, vol. 40, no. 1, Sep. 2020, Art. no. 4. [Online]. Available: https://doi.org/10.1145/3412360

[41] F. Losasso, F. Gibou, and R. Fedkiw, "Simulating water and smoke with an octree data structure," *ACM Trans. Graph.*, vol. 23, pp. 457–462, 2004.

[42] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis, "SPGrid: A sparse paged grid structure applied to adaptive smoke simulation," *ACM Trans. Graph.*, vol. 33, no. 6, 2014, Art. no. 205.

[43] P. Clausen, M. Wicke, J. R. Shewchuk, and J. F. O'brien, "Simulating liquids and solid-liquid interactions with Lagrangian meshes," *ACM Trans. Graph.*, vol. 32, no. 2, 2013, Art. no. 17.

[44] R. Ando, N. Thürey, and C. Wojtan, "Highly adaptive liquid simulations on tetrahedral meshes," *ACM Trans. Graph.*, vol. 32, no. 4, 2013, Art. no. 103.

[45] M. Becker and M. Teschner, "Weakly compressible SPH for free surface flows," in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*, 2007, pp. 209–217.

[46] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible SPH," *ACM Trans. Graph.*, vol. 28, no. 3, 2009, Art. no. 40.

[47] N. Akinci, M. Ihmsen, G. Akinci, B. Solenthaler, and M. Teschner, "Versatile rigid-fluid coupling for incompressible SPH," *ACM Trans. Graph.*, vol. 31, no. 4, 2012, Art. no. 62.

[48] R. Winchenbach, H. Hochstetter, and A. Kolb, "Infinite continuous adaptivity for incompressible SPH," *ACM Trans. Graph.*, vol. 36, no. 4, 2017, Art. no. 102.

[49] F. de Goes, C. Wallez, J. Huang, D. Pavlov, and M. Desbrun, "Power particles: An incompressible fluid solver based on power diagrams," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 50–51, 2015.

[50] J. Bender and D. Koschier, "Divergence-free SPH for incompressible and viscous fluids," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 3, pp. 1193–1206, Mar. 2017.

[51] S. Band, C. Gissler, and M. Teschner, "Moving least squares boundaries for SPH fluids," in *Proc. 13th Workshop Virt. Reality Interact. Physical Simulations*, 2017, pp. 21–28.

[52] S. Band, C. Gissler, M. Ihmsen, J. Cornelis, A. Peer, and M. Teschner, "Pressure boundaries for implicit incompressible SPH," *ACM Trans. Graph.*, vol. 37, no. 2, pp. 14:1–14:11, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3180486

[53] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, "The affine particle-in-cell method," *ACM Trans. Graph.*, vol. 34, no. 4, 2015, Art. no. 51.

[54] X. Zhang, M. Li, and R. Bridson, "Resolving fluid boundary layers with particle strength exchange and weak adaptivity," *ACM Trans. Graph.*, vol. 35, no. 4, 2016, Art. no. 76.

[55] Y. Zhu and R. Bridson, "Animating sand as a fluid," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 965–972, 2005.

[56] M. Geier, A. Greiner, and J. G. Korvink, "Cascaded digital lattice Boltzmann automata for high reynolds number flow," *Phys. Rev. E*, vol. 73, no. 6, 2006, Art. no. 066705.

[57] D. Lycett-Brown, K. H. Luo, R. Liu, and P. Lv, "Binary droplet collision simulations by a multiphase cascaded lattice Boltzmann method," *Phys. Fluids*, vol. 26, no. 2, 2014, Art. no. 023303.

[58] M. Geier, A. Greiner, and J. G. Korvink, "A factorized central moment lattice Boltzmann method," *Eur. Phys. J. Special Topics*, vol. 171, no. 1, pp. 55–61, 2009.

[59] X. Shan *et al.*, "Central-moment-based Galilean-invariant multiple-relaxation-time collision model," *Physical Rev. E*, vol. 100, no. 4, 2019, Art. no. 043308.

[60] A. De Rosis, "Nonorthogonal central-moments-based lattice Boltzmann scheme in three dimensions," *Physical Rev. E*, vol. 95, no. 1, 2017, Art. no. 013310.

[61] M. Rohde, D. Kandhai, J. Derksen, and H. Van den Akker, "Improved bounce-back methods for no-slip walls in lattice-Boltzmann schemes: Theory and simulations," *Physical Rev. E*, vol. 67, no. 6, 2003, Art. no. 066703.

[62] M. Sbragaglia and S. Succi, "Analytical calculation of slip flow in lattice Boltzmann models with kinetic boundary conditions," *Physics Fluids*, vol. 17, no. 9, 2005, Art. no. 093602.

[63] J. C. Verschaeve and B. Müller, "A curved no-slip boundary condition for the lattice Boltzmann method," *J. Comput. Phys.*, vol. 229, no. 19, pp. 6781–6803, 2010.

[64] R. Mei, L.-S. Luo, and W. Shyy, "An accurate curved boundary treatment in the lattice Boltzmann method," *J. Comput. Phys.*, vol. 155, no. 2, pp. 307–330, 1999.

[65] Z.-G. Feng and E. E. Michaelides, "The immersed boundary-lattice Boltzmann method for solving fluid-particles interaction problems," *J. Comput. Phys.*, vol. 195, no. 2, pp. 602–628, 2004.

[66] J. Wu and C. Shu, "An improved immersed boundary-lattice Boltzmann method for simulating three-dimensional incompressible flows," *J. Comput. Phys.*, vol. 229, no. 13, pp. 5022–5042, 2010.

[67] Y. Sato, T. Hino, and K. Ohashi, "Parallelization of an unstructured Navier-Stokes solver using a multi-color ordering method for openmp," *Comput. Fluids*, vol. 88, pp. 496–509, 2013.

[68] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable hybrid MPI/OpenMP unstructured finite element model," *Comput. Fluids*, vol. 110, pp. 227–234, 2015.

[69] O. Mashayekhi, C. Shah, H. Qu, A. Lim, and P. Levis, "Automatically distributing Eulerian and hybrid fluid simulations in the cloud," *ACM Trans. Graph.*, vol. 37, no. 2, 2018, Art. no. 24.

[70] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler solver using commodity graphics hardware," in *Proc. 46th AIAA Aerosp. Sci. Meeting Exhibit*, 2008, Art. no. 607.

[71] Y. Liu, X. Liu, and E. Wu, "Real-time 3D fluid simulation on GPU with complex obstacles," in *Proc. 12th Pacific Conf. Comput. Graph. Appl.*, 2004, pp. 247–256.

[72] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, 2003.

[73] CUDA, "Parallel computing platform," 2021. [Online]. Available: https://developer.nvidia.com/cuda-zone

[74] cuSPARSE, "Sparse linear algebra," 2021. [Online]. Available: https://developer.nvidia.com/cusparse

[75] J. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *Proc. 47th AIAA Aerosp. Sci. Meeting Including New Horizons Forum Aerosp. Expo.*, 2009, Art. no. 758.

[76] M. Griebel and P. Zaspel, "A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations," *Comput. Sci.-Res. Develop.*, vol. 25, no. 1–2, pp. 65–73, 2010.

[77] R. Henniger, D. Obrist, and L. Kleiser, "High-order accurate solution of the incompressible Navier-Stokes equations on massively parallel computers," *J. Comput. Phys.*, vol. 229, no. 10, pp. 3543–3572, 2010.

[78] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids," in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*, 2010, pp. 65–74.

[79] M. Gao *et al.* "GPU optimization of material point methods," *ACM Trans. Graph.*, vol. 37, 2018, Art. no. 254.

[80] K. Wu, N. Truong, C. Yuksel, and R. Hoetzlein, "Fast fluid simulations with sparse volumes on the GPU," in *Computer Graphics Forum*, Hoboken, NJ, USA: Wiley, 2018, pp. 157–167.

[81] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: A language for high-performance computation on spatially sparse data structures," *ACM Trans. Graph.*, vol. 38, no. 6, pp. 1–16, 2019.

[82] G. Alfonsi, S. A. Ciliberti, M. Mancini, and L. Primavera, "Performances of Navier-Stokes solver on a hybrid CPU/GPU computing system," in *Proc. Int. Conf. Parallel Comput. Technol.*, 2011, pp. 404–416.

[83] Y. Wang, M. Baboulin, K. Rupp, O. Le Maître, and Y. Fraigneau, "Solving 3D incompressible Navier-Stokes equations on hybrid CPU/GPU systems," in *Proc. High Perform. Comput. Symp.*, 2014, Art. no. 12.

[84] S. Posey, "Considerations for GPU acceleration of parallel CFD," *Procedia Eng.*, vol. 61, pp. 388–391, 2013.

[85] W. Li, X. Wei, and A. Kaufman, "Implementing lattice Boltzmann computation on graphics hardware," *Vis. Comput.*, vol. 19, no. 7–8, pp. 444–456, 2003.

[86] Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman, "Flow simulation with locally-refined LBM," in *Proc. Symp. Interactive 3D Graph. Games*, 2007, pp. 181–188.

[87] G. Wellein, T. Zeiser, G. Hager, and S. Donath, "On the single processor performance of simple lattice Boltzmann kernels," *Comput. Fluids*, vol. 35, no. 8–9, pp. 910–919, 2006.

[88] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, and J. Westerholm, "An efficient swap algorithm for the lattice Boltzmann method," *Comput. Phys. Commun.*, vol. 176, no. 3, pp. 200–210, 2007.

[89] N. Delbosc, J. L. Summers, A. Khan, N. Kapur, and C. J. Noakes, "Optimized implementation of the lattice Boltzmann method on a graphics processing unit towards real-time fluid simulation," *Comput. Math. Appl.*, vol. 67, no. 2, pp. 462–475, 2014.

[90] J. Tölke, "Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA," *Comput. Vis. Sci.*, vol. 13, no. 1, 2010, Art. no. 29.

[91] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating lattice Boltzmann fluid flow simulations using graphics processors," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 550–557.

[92] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on NVIDIA GPUs Using CUDA," *Advances Eng. Softw.*, vol. 42, no. 5, pp. 266–272, 2011.

[93] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras, "A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries," *Concurr. Comput., Pract. Exp.*, vol. 22, no. 1, pp. 1–14, 2010.

[94] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, "Multi-GPU implementation of the lattice Boltzmann method," *Comput. Math. Appl.*, vol. 65, no. 2, pp. 252–261, 2013.

[95] J. Myre, S. D. Walsh, D. Lilja, and M. O. Saar, "Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters," *Concurr. Comput., Pract. Exp.*, vol. 23, no. 4, pp. 332–350, 2011.

[96] S. Harris, *An Introduction to the Theory of the Boltzmann Equation*. Chelmsford, MA, USA: Courier Corporation, 2004.

[97] S. Girimaji, "Lattice Boltzmann method: Fundamentals and engineering applications with computer codes," *AIAA J.*, vol. 51, no. 1, pp. 278–279, 2013.

[98] C. Yuksel, "Sample elimination for generating Poisson disk sample sets," in *Computer Graphics Forum*, Hoboken, NJ, USA: Wiley, 2015, pp. 25–32.

[99] T. Krüger, F. Varnik, and D. Raabe, "Efficient and accurate simulations of deformable particles immersed in a fluid using a combined immersed boundary lattice Boltzmann finite element method," *Comput. Math. Appl.*, vol. 61, no. 12, pp. 3485–3505, 2011.

[100] K. Mattila, J. Hyväluoma, J. Timonen, and T. Rossi, "Comparison of implementations of the lattice-Boltzmann method," *Comput. Math. Appl.*, vol. 55, no. 7, pp. 1514–1524, 2008.

[101] P. Valero-Lara, F. D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier, "Accelerating fluid-solid simulations (lattice-Boltzmann & immersed-boundary) on heterogeneous architectures," *J. Comput. Sci.*, vol. 10, pp. 249–261, 2015.

[102] G. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," *Int. Bus. Mach. Company*, 1966.

[103] A. Heron and J. Adam, "Particle code optimization on vector computers," *J. Comput. Phys.*, vol. 85, no. 2, pp. 284–301, 1989.

[104] Jakob, "Mitsuba renderer," 2010. [Online]. Available: http://www.mitsuba-renderer.org/
[105] K. Suga, Y. Kuwata, K. Takashima, and R. Chikasue, "A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows," *Comput. Math. Appl.*, vol. 69, no. 6, pp. 518–529, 2015.
[106] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *J. Comput. Appl. Math.*, vol. 236, no. 15, pp. 3584–3590, 2012.

Yixin Chen received the BEng degree in computer science and technology from ShanghaiTech University, China, in 2019. She is currently working toward the PhD degree with the University of Toronto. After graduation, she worked as a research assistant with the School of Information Science and Technology (SIST), ShanghaiTech University. Her current interests include high-performance large-scale fluid simulations for computer graphics, computational design and fabrication, as well as machine learning for training robots in virtual environments.

Wei Li received the BSc degree in computer science from the Northwestern Polytechnical University, China, in 2015. He is currently working toward the PhD degree with the School of Information Science and Technology (SIST), ShanghaiTech University. He is interested in creating virtual worlds using computers, specifically in the areas of computer graphics, physically-based animation, computational physics, scientific computing, rendering and visualization, as well as robot learning.

Rui Fan received the BSc degree in computer science and mathematics from Caltech, in 2000, and the PhD degree in computer science from MIT, in 2008. He is currently an associate professor of computer science with the School of Information Science and Technology, ShanghaiTech University. Prior to joining ShanghaiTech, he was an assistant professor with Nanyang Technological University. His main research interests include parallel and distributed computing, as well as energy efficient computing and resource management. His work currently includes using parallel computing to accelerate machine learning, applying machine learning techniques to accelerate discrete algorithms, and developing parallel algorithms for large-scale bioinformatics problems and data mining.

Xiaopei Liu received the PhD degree in computer science and engineering from the Chinese University of Hong Kong, in 2010, and later worked as a research fellow with Nanyang Technological University. He is currently an assistant professor with the School of Information Science and Technology, ShanghaiTech University. His current research interests include computer graphics and physically-based simulation, computational physics, scientific visualization, machine learning and robotics, as well as distributed and parallel computing techniques and their applications in different domains.