Research Article

Real-Time Incompressible Fluid Simulation on the GPU

Xiao Nie,^{1,2} Leiting Chen,^{1,2} and Tao Xiang¹

¹School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China

²Provincial Key Laboratory of Digital Media, Chengdu 611731, China

Correspondence should be addressed to Xiao Nie; nixiao2008@gmail.com

Received 24 November 2014; Accepted 29 December 2014

Academic Editor: Manuel M. Oliveira

Copyright © 2015 Xiao Nie et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We present a parallel framework for simulating incompressible fluids with predictive-corrective incompressible smoothed particle hydrodynamics (PCISPH) on the GPU in real time. To this end, we propose an efficient GPU streaming pipeline to map the entire computational task onto the GPU, fully exploiting the massive computational power of state-of-the-art GPUs. In PCISPH-based simulations, neighbor search is the major performance obstacle because this process is performed several times at each time step. To eliminate this bottleneck, an efficient parallel sorting method for this time-consuming step is introduced. Moreover, we discuss several optimization techniques including using fast on-chip shared memory to avoid global memory bandwidth limitations and thus further improve performance on modern GPU hardware. With our framework, the realism of real-time fluid simulation is significantly improved since our method enforces incompressibility constraint which is typically ignored due to efficiency reason in previous GPU-based SPH methods. The performance results illustrate that our approach can efficiently simulate realistic incompressible fluid in real time and results in a speed-up factor of up to 23 on a high-end NVIDIA GPU in comparison to single-threaded CPU-based implementation.

1. Introduction

Physically based simulation methods have revolutionized special effects in computer games and interactive applications, creating incredible scenes consisting of fluids, cloth, rigid bodies, and others. Among these methods, *smoothed particle hydrodynamics* (SPH) is a well-known particle-based approach with a wide application for simulating many kinds of phenomena including water, fire, and gas. This paper focuses on simulating incompressible fluids for real-time visual effects.

Many studies have been devoted to real-time fluid simulation methods in recent years. However, these techniques suffer from visible compression artifacts. Although weakly compressible SPH (WCSPH) is the most widespread technique for pressure conservation in SPH based methods, the challenge with WCSPH is that to get sufficient realism requires very small time step. To solve this problem, PCISPH is introduced to enforce incompressibility with significantly larger time steps. While each iteration step costs more, the overall time required in each time step to achieve incompressibility is

less. Although this makes it a good candidate for simulating realistic fluid phenomenon, it is still too computationally intensive for the state-of-the-art CPU to put into real-time applications.

The modern GPU substantially outpaces its CPU counterpart in arithmetic throughput and memory bandwidth, making it the ideal processor to boost a vast variety of data parallel applications such as SPH-based fluid simulations. Originally GPU was designed for real-time graphics rendering applications. In the past decade, it has been established as one of the major parallel processors for computationally intensive tasks. Advances in massively parallel GPU architectures have opened doors to employing PCISPH-based methods for simulating realistic fluid phenomena in real time. Modern GPUs normally have thousands of compute cores which deliver tremendous computing power. PCISPH exposes a high degree of parallelism, making it a perfect candidate for state-of-the-art GPUs. In the SPH literature, several GPU-accelerated techniques have been proposed to speed up the time-consuming physics calculation. However,

FIGURE 1: Solid bunny scene. The water was sampled with 35 k fluid particles, and the solid bunny was sampled with 5 k boundary particles.

efficient GPU-based PCISPH solver for real-time incompressible fluids has not been presented.

In SPH-based fluid simulation, the physical attributes of one particle are typically integrated over its dynamically changing neighbors at every time step, and neighbor search is the computationally most expensive part of the simulation, especially when PCISPH is used since this algorithm requires searching neighboring particles multiple times per simulation step due to its iterative feature. Therefore, efficient neighbor search is critical for performance in PCISPH-based fluid simulations. In literature, acceleration data structures such as uniform grids and kd-trees are commonly used to improve the neighbor search performance. In our framework, uniform grid is adopted due to the single resolution of particles. To further utilize the spatial coherence of neighboring particles and temporal coherence of a particle between two subsequent physics steps in this spatial partitioning scheme, we introduce an efficient parallel sorting method for GPU-based neighbor search. The proposed technique is then compared with previous approaches from a performance point of view.

In this paper, we design an efficient GPU pipeline for PCISPH to make use of the substantial parallelism inherent in this algorithm. By implementing this pipeline using CUDA on modern GPUs, we achieve one order of magnitude improvement over a single-threaded CPU-based algorithm. Figure 1 illustrates our first example where the interaction of incompressible fluids and solid bunny is modelled using our GPU pipeline in real time.

Our main contributions are summarized as follows:

- (i) an efficient GPU streaming pipeline for real-time incompressible fluid simulation,
- (ii) a new parallel sorting method for efficient SPH neighbor search,
- (iii) several CUDA optimization techniques for the proposed method, such as using fast on-chip shared memory, constant cache, and read-only cache to reduce global memory data access.

Our paper is organized as follows. After related work is reviewed in Section 2, we give a brief overview of the predictive corrective incompressible SPH (PCISPH) algorithm in Section 3. Our GPU simulation framework and optimization techniques are introduced in Section 4, followed by the visual and performance results in Section 5. Finally, this paper is concluded in Section 6.

2. Related Work

SPH is a mesh-free Lagrangian method which was originally invented for astrophysical applications [1, 2]. Muller et al. first introduced this technique to simulate compressible liquids [3]. Keiser et al. [4] proposed a unified SPH framework to handle interaction of compressible fluids and deforming solids. Solenthaler et al. [5] extended this unified particle model to animate melting and solidification phenomena.

Adams et al. [6] presented an adaptive sampling algorithm based on SPH to improve simulation efficiency without observably affecting the fluid flow behavior. Goswami and Pajarola [7] accelerated SPH fluid simulation by using a globally adaptive time discretization method and by saving computations on the inactive fluid particles. Goswami and Batty [8] proposed a regional time stepping technique for WSPH that can obtain a two times speedup over the traditional method.

Nevertheless, all these methods compute pressure from density using ideal gas equation which results in compressibility artifacts. To avoid spurious visual effects due to compressibility, many solutions have been presented. Cummins and Rudman [9] presented a pressure projection formulation for SPH fluids, where a pressure Poisson equation (PPE) was used to solve pressure. Premože et al. [10] introduced incompressible SPH (ISPH) method to the graphics community. Although this approach allows for large time steps, the PPE pressure computation is extremely computationally expensive per time step. To avoid time-consuming Poisson equation solvers, Becker and Teschner [11] introduced Tait equation [12] to computer graphics to deal with nearly incompressible flows. The cost per simulation step of this method is very low, but it requires small times due to high stiffness to enforce incompressibility. To combine the advantages of both ISPH and WSPH, that is, low cost per step and large time steps, Solenthaler and Pajarola [13] proposed PCISPH method which can handle typical incompressible scenarios with small fluid density deviations of down to 0.01% and comparatively large time steps. Ihmsen et al. [14] extended PCISPH to simulate incompressible fluids with complex boundaries. An extension of PCISPH for realizing two-way coupling with rigid bodies has been proposed in [15]. Recently, He et al. [16] presented a local Poisson SPH (LPSPH) approach based on equation of state to enforce incompressibility. Ihmsen et al. [17] proposed a formulation of the projection method which combines an SPH form of the pressure force and an SPH discretization of the continuity equation to obtain a discretized form of the PPE. Cornelis et al. [18] further integrated this method with fluid-implicit-particle (FLIP) to solve mass preservation issue in FLIP and efficiency issue in SPH.

In the past years, many solutions have been presented to take advantage of GPUs to simulate physically based effects such as SPH fluids that are computationally intensive [19]. Amada et al. [20] proposed a GPU-assisted approach to solve SPH equations, in which case the physical properties calculations were carried out on the GPU, but the neighbor map creation was still executed on the CPU. Hegeman et al. [21] used a dynamic quadtree structure to accelerate the neighbor queries in SPH-based fluid simulations in 2D scenarios. Zhang et al. [22] presented an adaptive sampling algorithm for simulating and visualizing weakly compressible fluid on the GPU, which used RGBA channels to store fluid particle indices and mapped uniform grid onto a 2D texture.

However, the aforementioned techniques are difficult to implement without expert knowledge of the graphics API even with a higher-level programming environment. This has been changed since the appearance of NVIDIA's CUDA.

Since its introduction in 2006, many research works have been carried out with this general-purpose parallel programming model. Harada et al. [23] fully exploited the massive computational power of programmable graphics hardware by developing a method that executes all calculations including neighbor search entirely on the GPU. Harada [24] presented a method using CUDA instead of graphics APIs, which can more easily make use of multiple processors with distributed memory to accelerate particle-based fluid simulation. Green [25] adopted an index sorting approach to improve the performance of neighbor search on the GPU. Later, Goswami et al. [26] utilized shared memory to further optimize the CUDA kernels where neighboring particles information is recalculated instead of being stored on the device. Zhang et al. [27] used multi-GPUs to improve the efficiency of weakly compressible fluid simulation and to overcome the device memory limitations of a single GPU.

3. Background

SPH is a mesh-free technique in which fluid volume is discretized into a finite set of particles. The fluid properties at an arbitrary point are accumulated using its neighboring particles. That is,

$$A_i = \sum_j \frac{m_j}{\rho_j} A_j \mathbf{W}_{ij}(h), \quad (1)$$

where A_i is the physical property at particle i . m_j , ρ_j , A_j are mass, density, and physical property at particle j . $\mathbf{W}_{ij}(h)$ is the smoothing kernel in the support radius h of particle i . The density of fluid particle i can be calculated by substituting ρ into (1) as

$$\rho_i = \sum_j m_j \mathbf{W}_{ij}(h). \quad (2)$$

Similarly, we can obtain the pressure force for an arbitrary particle i :

$$\mathbf{F}_i^{\text{pressure}} = -m_i \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla \mathbf{W}_{ij}(h) \quad (3)$$

and its viscosity forces:

$$\mathbf{F}_i^{\text{viscosity}} = \mu \sum_j \frac{m_j}{\rho_j} (\mathbf{v}_j - \mathbf{v}_i) \nabla^2 \mathbf{W}(h), \quad (4)$$

where $\mathbf{v}_i$ and $\mathbf{v}_j$ are the velocities of particle i and its neighbor j .

However, unrealistic compressibility artifacts are produced when simulating fluid using standard SPH method, since the ideal gas state equation is used to relate pressure to density. To avoid this problem, an iterative equation of state solver PCISPH (see Algorithm 1) is introduced where a predictor-corrector technique is applied to iteratively calculate fluid pressures that enforce incompressibility. At the beginning of each time step, neighbor search is first executed,

```

while simulating do
  foreach particle  $i$  do
    search neighbors  $N_i(t)$ 
  foreach particle  $i$  do
    compute viscosity force  $\mathbf{F}_i^{\text{viscosity}}$  (e.g. (4))
    compute gravity force  $\mathbf{F}_i^{\text{gravity}} = m_i \mathbf{g}$ 
    compute other forces  $\mathbf{F}_i^{\text{others}}(t)$ 
    initialize pressure  $p(t) = 0.0$ 
    initialize pressure force  $\mathbf{F}_i^{\text{pressure}}(t) = 0.0$ 
  while ( $\rho_{\text{err}}^*(t+1) > \eta$ ) do
    foreach particle  $i$  do
      predict velocity  $\mathbf{v}_i^*(t+1)$ 
      predict position  $\mathbf{x}_i^*(t+1)$ 
    foreach particle  $i$  do
      predict density  $\rho_i^*(t+1)$  (e.g. (2))
      predict density variation  $\rho_{\text{err}_i}^*(t+1) = \rho_i^*(t+1) - \rho_0$ 
      update pressure  $p_i(t)$  (e.g. (5))
    foreach particle  $i$  do
      compute pressure force  $\mathbf{F}_i^{\text{pressure}}$  (e.g. (3))
  foreach particle  $i$  do
    compute total force  $\mathbf{F}_i^{\text{total}}(t)$  (e.g. (7))
    update velocity  $\mathbf{v}_i(t + \Delta t)$  (e.g. (8))
    update position  $\mathbf{x}_i(t + \Delta t)$  (e.g. (9))

```

ALGORITHM 1: CPU-based PCISPH algorithm.

followed by calculating all forces except for pressure force, which will be used to predict positions and velocities of fluid particles. Afterwards, the prediction-correction loop is performed to propagate density fluctuations through the fluid and update pressure and pressure forces until the desired density variation constraint is satisfied; that is, $\rho_{\text{err}}^*(t+1)$ is smaller than a user defined threshold η . In each iteration, the pressure of particle i is updated according to

$$p_i(t) += \delta \rho_{\text{err}_i}^*(t+1), \quad (5)$$

where δ is

$$\delta = \frac{\rho_0^2}{2\Delta t^2 m^2 \sum_j (\nabla W_{ij} \cdot \nabla W_{ij})}. \quad (6)$$

The pressure force of particle i is updated according to (3). Finally, time integration is executed to advance the positions and velocities of all particles using the total force exerted on each particle:

$$\mathbf{F}_i^{\text{total}} = \mathbf{F}_i^{\text{pressure}} + \mathbf{F}_i^{\text{viscosity}} + \mathbf{F}_i^{\text{gravity}} + \mathbf{F}_i^{\text{others}}. \quad (7)$$

4. Our GPU Simulation Framework

PCISPH-based fluid simulator requires intensive numerical computations. Therefore, it is important to design an efficient strategy to map all the physical simulation steps onto the GPU to improve its performance. In this section, we introduce a GPU-friendly particle data representation and a parallel sorting technique for fast neighbour search and develop an incompressible fluid simulation pipeline fully on the GPU to take advantage of its massively parallel processing power.

4.1. Particle Data Representation. In CPU-based PCISPH method, a single particle is represented as follows:

Struct Particle

```

{
  float position[3];
  float velocity[3];
  float predicted_position[3];
  float external_force[3];
  float correction_pressure_force[3];
  float correction_pressure;
  float predicted_density;
  float pressure;
  int type;
  int index;
}

```

Thus, the whole particle sets are defined as

Particle particleArray[N],

where N is the total number of particles. If we access each component of Particle concurrently, then AoS makes sense because, in this situation, successive reads of all components of each particle are contiguous. However, we observe that not all physical attributes are accessed at the same time in our simulation; for example, fluid density evaluation only needs to read particle position data. In other words, scattered memory access pattern occurs. In such a simulation, it is

generally better to store fluid particle properties in structure of arrays (SoA) instead of array of structures (AoS) to achieve a higher GPU memory bandwidth. This is due to the reason that a better coalesced memory access can be satisfied which results in optimum performance.

Therefore, in our implementation we patch fluid particle data in a SoA manner:

```

Struct Particle
{
    float4 position[N];
    float4 velocity[N];
    float4 predicted_position[N];
    float4 external_force[N];
    float4 correction_pressure_force[N];
    float correction_pressure[N];
    float predicted_density[N];
    float pressure[N];
    int type[N];
    int index[N];
}

```

For CUDA architecture, this means that memory coalescing and successive data are contained within the same cache line, leading to optimal global memory bandwidth and high cache-hit rate. Note that we apply padding strategy in our particle data structures; that is, we add a fourth dummy element to each 3-dimensional vector to ensure 4-byte alignment which is also critical for achieving high performance on the GPU.

4.2. Parallel Sorting Method for GPU-Based Neighbor Search. In PCISPH, the particle neighbor search step is performed many times in each physics update. Thus, it is critical for improving the total performance of the simulation system that we can efficiently query particle neighbors. As with most CPU-based and GPU-based SPH methods, we use 3D uniform grid data structure to accelerate neighbor search process since all particles have the same radius in our framework.

In uniform grid, each fluid particle is assigned to one grid cell according to its spatial location. To determine the neighborhood of one particle, we only need to search for particles that locate in the same grid cell or in one of the other 26 neighboring grid cells. In other words, 27 grid cells need to be queried for every particle. Although uniform grid is one of the most widely used spatial subdivision techniques on the CPU side, the parallel construction of it on the GPU is not quite straightforward due to race conditions of particle insertion. To avoid memory conflicts on the parallel architectures such as GPU, index sort [25] is used. This approach sorts particles using parallel radix sort, which makes particles in the same grid cell also close in GPU memory. In this way, memory coherence on the CUDA-capable devices can be achieved to some extent. However, it still has the weakness that fluid

particles that are spatially close are not necessarily close in GPU memory since grid cells are ordered on the basis of x -dimensional, y -dimensional, and z -dimensional position one after the other. This cannot fully utilize the modern GPU architectures that have a hierarchical memory system which uses caches to work as a fast data store.

To further improve the total memory bandwidth, Z -index sort [26, 28] is used to sort particles so that the memory reads are sequential and coherent per thread. This space filling Z -curve has a good spatial locality feature for GPU caches, leading to a high cache-hit rate. However, we observe that the radix sort is done on keys and these keys are the grid cell IDs that we are sorting, and the particle positions in each grid cell are irrelevant. This means that we only need to sort grid cells instead of every particle which can avoid many duplicate key sortings. Based on this observation, we present a new sorting method that performs 1-radix sort on grid cells instead of n -radix sort on digits. To this end, we change the size of the radix to match the number of grid cells. Thus, we can sort on the grid cells directly instead of sorting on the first key, the second key, and so on. Here we use prefix sum to figure out how many fluid particles are in each cell and thus the memory layout in the global memory. To count the number of particles in each grid cell, we use the atomic add operations. After this step, we can have the offsets of a particle from the start of its cell. At this point, we have both the memory location of each grid cell and the offset of each particle, so all particles can be put into the updated location in one step in parallel. In the context of GPU-accelerated SPH fluids, this means that a further performance improvement can be achieved.

4.3. GPU Streaming Pipeline. In this section, we present our GPU streaming pipeline for incompressible fluid simulation with PCISPH (see Figures 2 and 3).

In order to fully exploit the parallel capabilities of modern GPU, we map all physical calculation steps to CUDA kernels and particle data to streaming data representation. As illustrated in Figures 2 and 3, we use the following streaming data representations.

- (1) Position streams and velocity streams: these streams describe the current state of the fluid under simulation. We use double buffers for each stream; for example, position stream 0 is used to store old positions that are read-only, while position stream 1 represents new positions that are writable. At the end of each physics update, these two streams reverse their role.
- (2) External force stream: the stream contains the sum of viscosity force, surface tension force, boundary force, gravity force, and other forces except internal pressure force. This external force information will be used to generate predicted position stream for the fluid particles involved in the prediction-correction loop.
- (3) Predicted position stream: the stream will be used to generate corrected pressure, predicted density, and density error of each particle.

FIGURE 2: Our GPU streaming pipeline for PCISPH. All the particle data are represented as GPU streams (shown in parallelograms). These streams are operated by different CUDA kernels (shown in rectangular boxes).

FIGURE 3: CUDA kernels and streaming data for prediction-correction loop.

- (4) Predicted density stream: the stream will be used to predict density variation and corrected pressure.
- (5) Corrected pressure stream: the stream will be used to update pressure for the pressure force calculation.
- (6) Corrected pressure force stream: the stream will be used for the next iteration or the time integration step.

- (7) Density error stream: the stream will be used to generate loop end flag that will be used to check the exit condition of the prediction-correction loop.

The different computational stages corresponding to particle sorting, external force calculation, and various steps of prediction-correction loop and time integration are

abstracted as CUDA kernels and performed in parallel on the GPU compute cores.

- (1) Parallel sorting kernel: we use the method described in Section 4.2 to sort fluid particles at the beginning of each physics update.
- (2) External force calculation kernel: in this kernel, we accumulate each particle's viscosity force, surface tension force, and boundary force using its position and velocity information, as well as the gravity force.
- (3) Position prediction kernel: at the start of each prediction-correction loop, we use this kernel to predict particle position. The collision with boundaries should be also handled in the kernel.
- (4) Predicted density and pressure calculation kernel: in this kernel, predicted density is accumulated according to (2) using previously predicted position information.
- (5) Corrected pressure force calculation kernel: this kernel computes corrected pressure force according to (3) using the predicted density and pressure value.
- (6) Maximum predicted density calculation kernel: we use parallel reduction in this kernel to get the maximum predicted density which will be used to control the loop end flag.
- (7) Time integration kernel: we use symplectic Euler integration scheme, which is a semi-implicit method that uses explicit Euler method to update velocity but uses the newly updated velocity instead of the old value to calculate position:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t \frac{\mathbf{F}_i^{\text{total}}(t)}{m_i}, \quad (8)$$

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t), \quad (9)$$

where $\mathbf{v}_i(t + \Delta t)$ and $\mathbf{x}_i(t + \Delta t)$ are the updated velocity and updated position of a fluid particle. $\mathbf{x}$ is the position, Δt is the time step, $\mathbf{F}$ is the force, and m is the mass of the particle.

- (8) Indices calculation kernel: at the end of each frame, we update particle index in this kernel that will be used for particle sorting at the beginning of the next frame.

4.4. Optimization Techniques. To achieve a better performance on the GPU, we use several optimization techniques. Double-buffering techniques combined with texture memory/read-only data cache are used to read and update position and velocity values of fluid particles. To be specific, at each physical computation step, we first obtain previous attributes through CUDA texture memory/read-only data cache from one of the double buffers. Then, we use these data to calculate their new values and write into the other one of the double buffers. After that we switch these two buffers when necessary. For constant parameters such as

TABLE 1: Two different GPUs are used for testing our parallel incompressible fluid simulation framework.

GPU	GTX 480	GTX 780
Stream processors	480	2304
Memory clock	3696 MHz	6008 MHz
Core clock	700 MHz	863–902 MHz
Memory capacity	1.5 GB	3.0 GB
GPU architecture	Fermi GF100	Kepler GK110

mass, support radius, viscosity constant, gravity constant, and other constant properties, we utilize CUDA's constant memory to store them for efficient data access, since constant cache is extremely effective due to the massive reuse of these attributes, increasing the compute to memory access ratio. This also helps to boost global memory bandwidth.

In SPH, the physical attributes at one particle such as density, pressure force, and viscosity force are approximately computed with a set of attributes at its neighboring particles. To take advantage of the fast on-chip memory of GPU, shared memory is also used to load data which are shared by neighboring particles, thus reducing the global memory access request. After parallel sorting, these neighboring particle data are most likely close in global memory which further improves performance due to the coalesced memory access pattern when loading data from global memory to shared memory.

As for our shared memory implementation, each CUDA thread in a block is assigned to one fluid particle i , and this thread copies all particles in the 27 neighboring grid cells of i into this thread's shared memory. Each CUDA thread outputs the updated attribute of one fluid particle. To save memory, we make use of the existing space as much as possible. For example, we use two arrays to store key-value pair for particle's z -index value and its index in the array: one is used to store both the input unsorted array and output sorted array; the other one is used as an auxiliary array to allow ping pong computation. This auxiliary array is also used for CUDA block counts generations, which allows for simulating large scenes with more particles with the same hardware.

5. Results

In this section, we give a description of our GPU implementation and make performance comparisons with their CPU-based counterparts on several benchmarks.

5.1. Implementation. We have tested our parallel framework on an Intel Core i7-3770 with 3.4 GHz and two different commodity GPUs: a NVIDIA GeForce GTX 480 (Fermi architecture with compute capability 2.0) and a NVIDIA GeForce GTX 780 (Kepler architecture with compute capability 3.5). The parameters of these two GPUs are shown in Table 1. For these two NVIDIA GPUs, we used CUDA Toolkit 6.5/Visual Studio 2010/Windows 7 Ultimate 64-bit as the development environment.

FIGURE 4: CBD (30 k particles). The water realistically bounces around the glass box. (a) Rendered with OpenGL. (b) Rendered with Pov-Ray.

FIGURE 5: Bunny drop (50 k particles). (a) Rendered with OpenGL. (b) Rendered with Pov-Ray.

In our implementation, fluid surface reconstruction technique similar to [5] was used, and solid boundary particles were sampled using [29]. For all simulations, we set the time step to 0.003 s, the fluid particle radius to $r = 0.0375$ m, the rest distance between initial particles to $2r$, the support radius to $h = 4r$, the fluid rest density to 1000 kg/m^3 , and the density fluctuation threshold to 1%.

5.2. Benchmarks. To test the performance of our framework, we used four different benchmarks. Each benchmark is tested on both the CPU and the GPU.

- (1) Solid bunny scene: in Figure 1, we simulated a bunny scene, in which water flows around it and finally calms down. This figure demonstrates that our GPU framework can realistically simulate incompressible fluids with complex boundaries in real time.
- (2) CBD scene: Figure 4 is a classical dam-break benchmark.
- (3) Bunny drop scene: in Figure 5, we simulated a scenario with a water bunny dropping into the pool.

- (4) Two dam breaks' scene: in Figure 6, we simulated a scene, in which two corner breaking dams collide and generate sprays and then converge into a water pool.

For all benchmarks, our framework can simulate incompressible fluids accurately and robustly in real time on the GPU.

5.3. Performance Comparisons. In this section, we compare the performance of our GPU framework with the-state-of-the-art CPU-based implementations, that is, single-threaded PCISPH implementation [13] and its multithreaded version [30].

Table 2 highlights the performance of our GPU framework and CPU-based implementations on four different benchmarks from which we can see that the speedups of our GPU implementation on GTX 480 compared with its single-threaded counterpart were 11.4 for CBD scene, 9.4 for solid bunny, 11.2 for bunny drop, and 6.6 for two dam breaks. These speedups are even higher on GTX 780 which was 19.5 for CBD scene, 18.2 for solid bunny, 23.2 for bunny drop, and 16.8 for two dam breaks.

FIGURE 6: Two dam breaks (60 k particles). (a) Rendered with OpenGL. (b) Rendered with Pov-Ray.

TABLE 2: Performance results: this table shows the average frame rates of our framework on the CPU and two different generations of GPUs (the performance is measured using average frames per second (fps) over 3 k time steps).

Scenes	CPU-based methods (fps)		Our method (fps)	
	Single-threaded Solenthaler [13]	Multithreaded Solenthaler [30] (speedup)	GTX 480 (speedup)	GTX 780 (speedup)
CBD	3.49	5.75 (1.6x)	39.84 (11.4x)	67.98 (19.5x)
Solid bunny	2.48	4.18 (1.7x)	23.29 (9.4x)	45.21 (18.2x)
Bunny drop	1.78	3.58 (2.0x)	19.91 (11.2x)	41.24 (23.2x)
Two dam breaks	2.07	3.31 (1.6x)	13.73 (6.6x)	34.68 (16.8x)

Figure 7 is the graphical illustration of the performance results presented in Table 2. From this figure, we can notice that our GPU-based incompressible fluid simulation framework works well on both Fermi and Kepler architectures and is also scalable to current Maxwell and future architectures. In other words, our simulation pipeline can be used to exploit the large-scale parallel capabilities of current and future GPUs. Since our GPU-based pipeline will benefit from more compute cores and higher memory bandwidth, we can expect higher frame rates on future GPU architectures.

Figure 8 shows execution time ratios for each step of bunny drop scene, that is, parallel sorting, block generation, external force calculation, prediction-correction loop, time integration, and index calculation. From the figure, we can see that the prediction-correction loop is the most time-consuming stage in the whole simulation pipeline, which occupies 78.45% of the total computational time. The second most computationally expensive stage is the external force

calculation which has a percentage of 17.13%. These two stages consume the vast majority of computing resources. This is because these two stages contain time-consuming neighbor search, especially in the prediction-correction loop which needs to perform neighbor search many times.

6. Conclusion and Future Work

We have presented an efficient GPU pipeline to utilize the inherent parallelism of the PCISPH algorithm to simulate incompressible fluids in real time. We further took advantage of shared memory on the GPU which has lower latency and much higher bandwidth than global memory. Moreover, we proposed an efficient parallel sorting method for SPH fluid simulations which can achieve a high cache hit rate and thus a higher simulation performance. Our parallel framework is able to accurately simulate incompressible fluids in real time on the GPU.

As for our future work, we will extend our pipeline to support multiple GPU system.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank Assistant Professor Dr. Prashant Goswami, Computer Graphics group at Blekinge Institute of Technology, for many insightful discussions. The authors are also thankful to the anonymous reviewers for their helpful comments. This project is supported by the National High Technology Research and Development Program of China (no. 2012AA011503), Project on the Integration of Industry, Education and Research of Guangdong Province (no. 2012B090600008), and the preresearch project

FIGURE 7: Average simulation frames rates over 3k time steps. ((a)–(d)) Performance of each benchmark for different implementations. (e) Performance for four different benchmarks on an NVIDIA GeForce GTX 780.

FIGURE 8: Execution time ratios. This figure highlights the performance for different stages of our GPU-based PCISPH pipeline (for the bunny drop scene on GTX 780).

(no. 51306050102). The armadillo model and bunny model are courtesy of the Stanford Computer Graphics Lab.

References

- [1] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," *The Astronomical Journal*, vol. 82, pp. 1013–1024, 1977.
- [2] J. J. Monaghan and R. A. Gingold, "Smoothed particle hydrodynamics—theory and application to nonspherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, no. 3, pp. 375–389, 1977.
- [3] M. Muller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159, 2003.
- [4] R. Keiser, B. Adams, D. Gasser, P. Bazzi, P. Dutré, and M. Gross, "A unified lagrangian approach to solid-fluid animation," in *Proceedings of the Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pp. 125–134, June 2005.
- [5] B. Solenthaler, J. Schläfli, and R. Pajarola, "A unified particle model for fluid-solid interactions," *Computer Animation and Virtual Worlds*, vol. 18, no. 1, pp. 69–82, 2007.
- [6] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas, "Adaptively sampled particle fluids," *ACM Transactions on Graphics*, vol. 26, no. 3, article 48, 2007.
- [7] P. Goswami and R. Pajarola, "Time adaptive approximate SPH," in *Proceedings of the 8th Workshop on Virtual Reality Interactions and Physical Simulations (VRIPHYS '11)*, pp. 19–28, December 2011.
- [8] P. Goswami and C. Batty, "Regional time stepping for SPH," in *Eurographics*, pp. 45–48, 2014.
- [9] S. J. Cummins and M. Rudman, "An SPH projection method," *Journal of Computational Physics*, vol. 152, no. 2, pp. 584–607, 1999.
- [10] S. Premoze, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker, "Particle-based simulation of fluids," *Computer Graphics Forum*, vol. 22, no. 3, pp. 401–410, 2003.
- [11] M. Becker and M. Teschner, "Weakly compressible SPH for free surface flows," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 209–217, 2007.
- [12] J. J. Monaghan, "Simulating free surface flows with SPH," *Journal of Computational Physics*, vol. 110, no. 2, pp. 399–406, 1994.
- [13] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible SPH," *ACM Transactions on Graphics*, vol. 28, no. 3, article 40, 2009.
- [14] M. Ihmsen, N. Akinci, M. Gissler, and M. Teschner, "Boundary handling and adaptive time-stepping for PCISPH," in *Proceedings of the 7th Workshop on Virtual Reality Interactions and Physical Simulations (VRIPHYS '10)*, pp. 79–88, November 2010.
- [15] N. Akinci, M. Ihmsen, B. Solenthaler, G. Akinci, and M. Teschner, "Versatile rigid-fluid coupling for incompressible SPH," *ACM Transactions on Graphics*, vol. 31, no. 4, article 62, 2012.
- [16] X. He, N. Liu, S. Li, H. Wang, and G. Wang, "Local poisson SPH for viscous incompressible fluids," *Computer Graphics Forum*, vol. 31, no. 6, pp. 1948–1958, 2012.
- [17] M. Ihmsen, J. Cornelis, B. Solenthaler, C. Horvath, and M. Teschner, "Implicit incompressible SPH," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 426–435, 2014.
- [18] J. Cornelis, M. Ihmsen, A. Peer, and M. Teschner, "IISPH-FLIP for incompressible fluids," in *Proceedings of the 35th Annual Conference of the European Association for Computer Graphics (Eurographics '14)*, Strasbourg, France, April 2014.
- [19] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, "SPH fluids in computer graphics," in *Eurographics 2014—State of the Art Reports*, The Eurographics Association, 2014.
- [20] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara, "Particle-based fluid simulation on GPU," in *Proceedings of ACM Workshop on General-Purpose Computing on Graphics Processors and (SIGGRAPH '04)*, Los Angeles, Calif, USA, August 2004.
- [21] K. Hegeman, N. A. Carr, and G. Miller, "Particle-based fluid simulation on the GPU," in *Computational Science—ICCS 2006*, vol. 3994 of *Lecture Notes in Computer Science*, pp. 228–235, Springer, 2006.
- [22] Y. Zhang, B. Solenthaler, and R. Pajarola, "Adaptive sampling and rendering of fluids on the GPU," in *Proceedings of IEEE/EG Symposium on Volume and Point-Based Graphics*, pp. 137–146, 2008.
- [23] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on GPUs," *Proceedings of Computer Graphics International*, pp. 63–70, 2007.
- [24] T. Harada, "Parallelizing particle-based simulation on multiple processors," in *Game Physics Pearls*, pp. 155–176, A K Peters/CRC Press, 2010.
- [25] S. Green, "Particle-based fluid simulation," in *Proceedings of the 2008 Game Developers Conference*, pp. 1–33, San Francisco, Calif, USA, 2008.
- [26] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, "Interactive SPH simulation and rendering on the GPU," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 55–64, 2010.

- [27] F. Zhang, X. Shen, X. Long, B. Zhao, and L. Hu, “Real-time particle fluid simulation with WCSPH,” in *Pacific Graphics Short Papers*, pp. 29–34, The Eurographics Association, 2012.
- [28] M. Ihmsen, N. Akinici, M. Becker, and M. Teschner, “A parallel SPH implementation on multi-core CPUs,” *Computer Graphics Forum*, vol. 30, no. 1, pp. 99–112, 2011.
- [29] M. Becker, H. Tessendorf, and M. Teschner, “Direct forcing for Lagrangian rigid-fluid coupling,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 493–503, 2009.
- [30] B. Solenthaler and R. Pajarola, “Performance comparison of parallel PCISPH and WCSPH,” Tech. Rep. IFI-2009.0, Department of Informatics, University of Zürich, 2009.