

# Novel Hierarchical Strategies for SPH-centric Algorithms on GPGPU

Kemeng Huang<sup>a</sup>, Zipeng Zhao<sup>a</sup>, Chen Li<sup>a</sup>, Changbo Wang<sup>\*a</sup>, Hong Qin<sup>b</sup>

<sup>a</sup>*School of Computer Science and Technology, East China Normal University, Shanghai, China*

<sup>b</sup>*Department of Computer Science, Stony Brook University, Stony Brook, USA*

---

## Abstract

This research further extends a existing state-of-the-art GPU framework of SPH simulation for better performance without any compromise of numerical accuracy and sacrifice of simulation detail. Towards this grand goal, we devise three new strategies. First, we design a novel hierarchical grid to decompose the simulation space, where we can locate particles in a more refined unit space. With this organization, particle distribution coherence in global memory is improved. As a result, global memory operations can be more efficient. Second, based on our well designed hierarchical grid, we propose a hierarchical neighbor search strategy for catering to the heterogeneous distribution of particles in cells, so we can search a neighbor particle in different-level grids. In a cell with dense particles, we can search particles in a higher-level grid whose unit space is more refined, which means we can narrow the search space for decreasing the access of false neighbor particles. In contrast, in a cell with sparse particles, we search particles in a lower-level grid whose unit space is relatively large, which means we can decrease the loop iterations when we search the entire neighbor cell. In this way, we can avoid the unnecessary overhead of loop iterations. After designing a reasonable neighbor search strategy, an efficient thread cooperation strategy can further improve the performance of our framework by realizing more potentials of GPGPU. The existing state-of-the-art method is concentrating on task assignment strategy for taking the full advantage of shared memory. The method decomposes particles in a cell into several tasks and then assigns a cooperation thread array (CTA) to each task. However, this method has not fully considered the uniformity of tasks belonging to the same cell, as these tasks always share a lot of particles with the same neighborhood. Finally, we propose a hierarchical task assignment strategy by merging such successive tasks into a larger task, which means we only load the same neighbor particles once and the corresponding CTAs are arranged to work together to handle the new task. Our method can greatly reduce the overload of neighbor particles and the overhead of neighbor search iterations. Through our comprehensive tests validated in practice, our work can exhibit 1.73x speedup when compared with other state-of-the-art GPGPU frameworks for SPH simulation.

**Keywords:** Smoothed Particle Hydrodynamics (SPH), Hierarchical Perpendicular Grid (HPG), Neighbor Search, Hierarchical Task Assignment

---

## 1. Introduction and Motivation

As a popular mesh-free Lagrangian-based method, Smooth Particle Hydrodynamics (SPH) has been widely employed in computational fluid dynamics, solid mechanics, and many other physics-based simulations pertinent to computer graphics. The underlying simulations requiring well detailed, large-scale, and complicated scenes will unavoidably increase a large number of particles, resulting in a significant time expense. Therefore, much research has been focusing on the performance improvement of SPH-centric algorithms in computer graphics.

To date, the rapid advancement of GPU's computational power, together with the software evolution of system development kits (e.g., CUDA), have led to increased interest in continuing to push forward the improved efficiency of GPU's parallel potential. Green [1] represents the earliest CUDA-based implementation of particle system using the uniform grid. A similar implementation, the earliest GPU based SPH [2] replaces the Cell Linked List (CLL) with Verlet list (VL) [3] [4] in 2010, which can exhibit one to two order-of-magnitude speedup compared with the equivalent CPU implementation. DualSPHysics [5], an open-source SPH solver, provides similar implementations

with several optimization strategies. Although these methods are efficient, they haven't taken the full advantage of thread cooperation and hierarchical memory in GPU. GpuSPHASE [6] [7], another open-source 2D SPH solver, presents a shared memory based implementation. As GpuSPHASE is a Verlet List based SPH algorithm, GpuSPHASE only loads the target particle data into shared memory, so the improvement of shared memory is limited. Later, Goswami et al. [8] first propose a full shared memory based method. However, without a reasonable shared memory loading strategy and thread cooperation strategy, their method does not exhibit better performance than DualSPHysics. Most recently, Huang et al. [9] proposes a novel general parallel framework (called named GpuSPHCTA in this paper), which successfully breaks the performance bottleneck by using thread cooperation and reasonable full shared memory loading strategy.

In this paper, our ambitious goal is to further improve the computational performance based on the GpuSPHCTA framework, and at the same time we also retain the computational accuracy without any sacrifice of involved numerical techniques. We first propose a hierarchical perpendicular uniform grid to ar-

range world-space particles, and our method could greatly improve the coherence of particle distribution and decrease the overhead of thread synchronization. Based on our novel grid method, we propose an efficient neighbor traversal algorithm, which greatly decreases the access of false neighbor particles. In addition, we introduce a hierarchical task schedule strategy, which utilizes different thread groups with different thread size to cater to the heterogeneous distribution of particles in the simulation space. Our primary contributions are summarized below.

- A novel hierarchical perpendicular grid for improving the coherence of particle distribution, with a goal to reduce the time cost of thread synchronization.
- A well designed hierarchical neighbor traversal algorithm, which can greatly decrease the access time of false neighbor particles.
- A hierarchical task schedule strategy, which can cater to the distribution of particles in the simulation space, with a goal to reduce the overload of neighbor particles.

## 2. Related Work

Accompany with the widely used of SPH in physically-based simulations, much research has been focusing on improving the performance of SPH-centric algorithms by GPU. Here we start by simply introducing the development of SPH in physically-based simulations, and several representative implementations of GPU based SPH algorithm will be provided.

### 2.1. SPH-based Simulations

SPH is firstly proposed for astrophysical problem [10], and then SPH is extended to several physically-based simulations in computational physics and computer graphics. The core idea of this method is discretizing continuous fields by means of particles and approximating continuous properties of particles by weighted interpolation over smoothed functions  $W(\mathbf{x}, h)$  [11]:

$$A(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{x}_i - \mathbf{x}_j, h), \quad (1)$$

where  $A(\mathbf{x}_i)$  is some consecutive attributes of the particle at position  $\mathbf{x}_i$ , such as force,  $m$  is the mass of particle and  $\rho$  is particle density. As shown in Fig. 1, smoothing radius  $h$  defines the influence space within which the contribution from the neighbor particles should be accumulated for target particle [9].

As a nature mass-conservation and flexible method, SPH has been widely used to simulate classical fluid flow [12] [13]. In addition, SPH can be also applied to solid simulations [14] [15], as well as the coupling and interaction of solid and fluid [16] [17]. In 2014, Ren et al. [18] extended WCSPH for simulating multiple fluid by a mixture model. And their method was then extended by Yan et al. [19] and Yang et al. [20] to cover the simulations of solid, including deformable solid and sand, as well as the transformation of different materials. In order to simulate incompressible fluid efficiently and stably, Bender et al. [21]

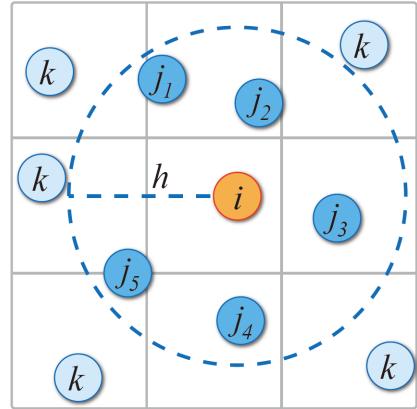


Figure 1: The circle denoted by  $i$  represents target particle. The circles denoted by  $j$  represent the true neighbor particles, while the circles denoted by  $k$  represent the false neighbor particles. In addition,  $h$  is the influence radius of target particle.

proposed DFSPH in 2017. In 2019, Koschier et al. [22] surveyed various aspects of SPH simulations with a documented tutorial.

### 2.2. GPU Implementations

Here we introduce GPU based SPH from three perspectives, early single GPU implementations, CUDA based single GPU implementations and multi-GPU implementations.

**Traditional Single GPU**, in 2004, Kipfer et al. [23] and Kolb et al. [24] proposed the earliest large particle system on GPU, which can simulate fluid with uncoupled particles. Kolb et al. [25] then proposed a GPU-based dynamic particle coupling method, which is the earliest implementation of SPH on GPU. Harada et al. [26] implemented the neighbor search process of SPH on GPUs. Their implementation was a fully GPU-based SPH method. As all of these methods were based on shader programs and occurring before the appearance of CUDA tool kit, it was difficult for these methods to well arrange the threads of GPU or manipulate the hierarchical memory resources. So the GPU's computational power was not fully utilized.

**CUDA based Single GPU**, in 2008, NVIDIA provided a CUDA based implementation of particle system on GPU [1], and a similar method [2] was provided in 2010, which is the earliest CUDA based implementation of SPH. In the same year, Goswami et al. [8] proposed the earliest full shared memory based method for SPH on GPU. However, their work did not have a better performance compared with DualSPHysics [5], a popular parallel framework for SPH, merged with several optimization strategies proposed by Domínguez et al. [27]. Xia et al. [28] proposed a new VL on GPU for SPH by using Quadtree neighbor searching instead of uniform grid. Although this method can reduce the redundant computational cost, it cannot take the full advantage of parallelization on GPU. GpuSPHASE [6] [7], an open-source 2D SPH solver, gave several optimization strategies. GpuSPHASE tried to use shared memory to reduce global memory access, but only target particles were loaded into shared memory, so the performance improvement of shared memory was quite limited. Ohno et al. [29] sub-

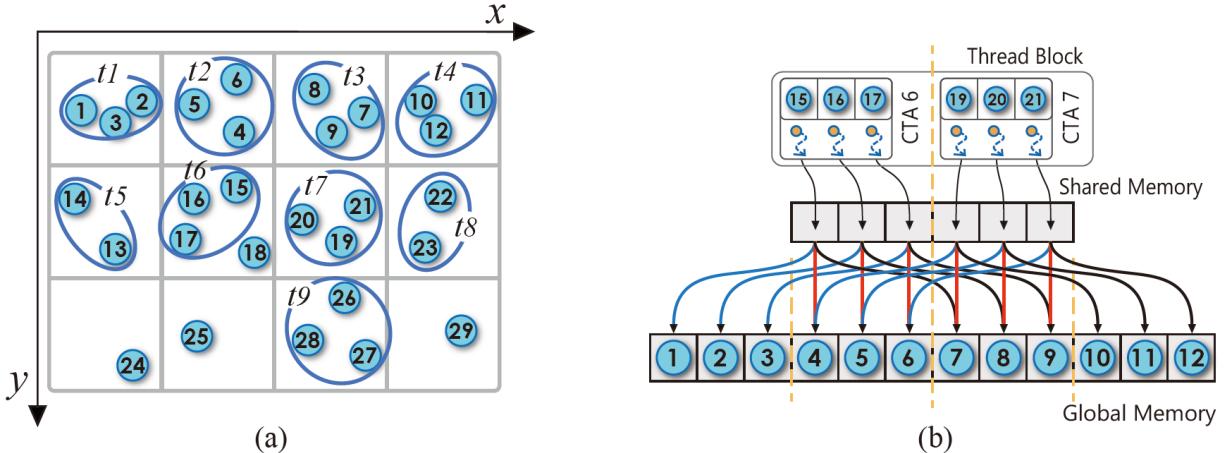


Figure 2: (a) gives the overview of task division strategy of GpuSPHCTA. The smallest circles represent particles, and the inside number represents the index of particle. All particles inside a big circle represent a task. (b) gives the overview of memory operations in a thread block. GpuSPHCTA launches a thread block to deal with two tasks, and assigns each task with a CTA. The dot lines with arrow represent threads. Arrows of solid lines represent the direction of memory requests. The lines, between shared memory and global memory, shown with different colors, represent the order of memory requests. Blue first, red second, and black third.

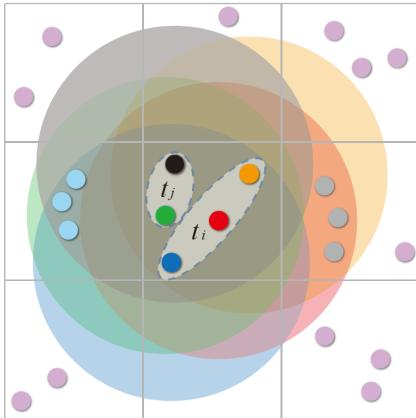


Figure 3: The overview of inconsistent neighbor search situation inside tasks. The relative small circles represent particles, and the bigger circles represent the influence regions of corresponding particles. The ellipses denoted with  $t$  represent tasks. Only neighbor particles inside the influence region are true neighbor particles.

divided the cells of uniform grid to build VL. Their method can improve the spatial locality of particles, so as to reduce the access of false neighbor particles. But this method involved high overhead of loop iterations, which can greatly reduce the performance of algorithm. In recent years, the performance of single-GPU based SPH has come to a bottleneck. In 2019, Huang et al. [9] successfully broke the performance bottleneck through the analysis of hardware architecture of GPU and taking relatively full use of computational power on GPU. This method can exhibit a good speedup, especially on dense particles, and keep the good performance on sparse particles by merging the traditional efficient method into their framework.

**Multi-GPUs**, in order to cater to the huge number of particles, distributing the computations among several GPUs is unavoidable. In 2011, Zhang et al. [30] implemented one of relatively early multi-GPUs SPH algorithm, using a dynamic

load balancing method. The similar method was used by Hu et al. [31] with some optimization strategies in load balancing method and communication strategy. In 2012, Rustico et al. [32] divided the simulation space into different GPUs and kept a minimal overlapping of sub-domains to ensure each particle can get all true neighbor particles. Their work was then extended [33] in 2014. In 2013, Valdez-Balderas et al. [34] also proposed a spatial decomposition technique based multi-GPUs SPH using Message Passing Interface (MPI). This work was extended by Domínguez et al. [35] with an improved MPI, the new work can handle simulations with more than one hundred million particles. In 2017, an advanced load balancing scheme for multi-GPUs SPH was proposed by Verma et al. [36], this method can also scale well for smaller amount of particles. In conclusion, research of multi-GPU SPH almost focused on split strategies and data load balancing methods between different GPUs during these years. Therefore, the main performance bottleneck is still depending on the performance of each GPU node.

### 3. The Foundation of Our Framework

Our framework is based on GpuSPHCTA [9], which is a state-of-the-art parallel framework. Through designing more reasonable grid method and task assignment strategy, we significantly improve the efficiency of SPH algorithms on GPU. We briefly introduce GpuSPHCTA in this section.

GpuSPHCTA is a uniform grid based method, which divides particles into several tasks according to the warp-size (32 in current GPUs) for each uniform grid cell and launches a CTA for each task. The core idea of GpuSPHCTA is reasonably manipulating threads and shared memory for declining global memory access in an efficient way. Fig. 2(a) gives an example of task division strategy in GpuSPHCTA (we assume the warp-size is 3 here). All particles in a task belong to a same cell, so these particles share the same neighbor cells, which means the

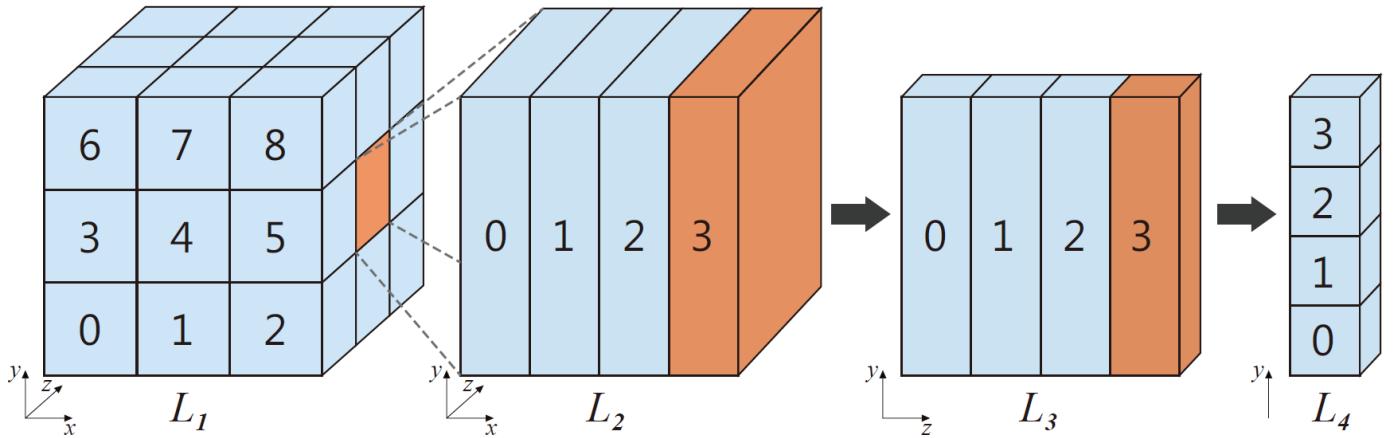


Figure 4: The overview of our hierarchical perpendicular grid. The cuboids colored with red represent target cuboids, which are subdivided into 4 cuboids of next grid level. And the number represent cuboid index in current grid level.

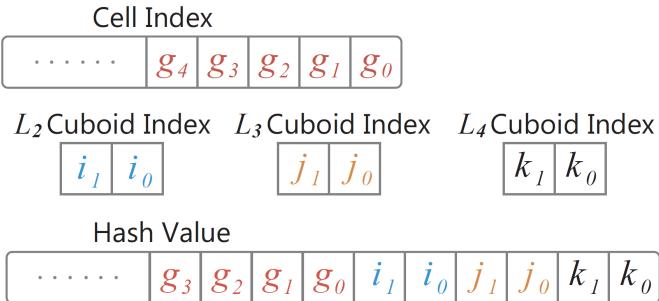


Figure 5: The overview of our encoding method. We first calculate cell index  $I$  for each particle. Then we calculate the sub-cuboid index in  $L_2$ ,  $L_3$  and  $L_4$  cuboid. Finally we calculate our new hash value through bit operations according to the cell index and three sub-cuboid index.

neighbor particles loaded by a target particle can be also used for other target particles in the same task. Fig. 2(b) gives the overview of the cooperation strategy of threads and the function of shared memory. First, the threads in a CTA consecutively load neighbor particles into shared memory; second, all target particles load neighbor particles from shared memory; finally, calculate attributes for each target particle. With this organization, massive global memory access can be avoided. In order to improve the cache hit rate, GpuSPHCTA arranges two CTAs as a CUDA thread block, which can be helpful for our task merging strategy.

In GpuSPHCTA, keeping the particles of a task in the same cell is a very important precondition. However, GpuSPHCTA can not promise the coherence of particle distribution inside a cell, because GpuSPHCTA involves atomic operations in the sorting for particles and the returned value of atomic operations in CUDA is random. Therefore, the particles in a cell are disordered, which might increase the overhead of thread synchronization and decline cache hit rate. In order to reduce such problem, we propose a novel hierarchical grid method, and we will detail it in Section 4.1. As GpuSPHCTA is a CLL based framework, each target particle has to load all particles

in neighbor cells for searching true neighbor particles, which can be optimized by reducing the search space in hierarchical grid, and we will detail it in Section 4.2. GpuSPHCTA is a single-level task framework, which means each CTA works separately. But we find that if two consecutive tasks belong to a same cell, such two tasks always share amount of same neighbor particles, which means loading neighbor particles for each task separately might result in unnecessary overloading. In order to reasonably avoid such overloading as much as possible, we proposed a hierarchical task assignment strategy to cater to the heterogeneous distribution of particles, and we will detail it in Section 4.3.

#### 4. Novel Hierarchical Strategy

First, we will illustrate our hierarchical perpendicular grid. Furthermore, we will introduce our hierarchical neighbor search strategy. Finally, we will detail our novel hierarchical task assignment strategy.

##### 4.1. Hierarchical Perpendicular Grid

Uniform grid is a simple but efficient method. Based on uniform grid, GpuSPHCTA and DualSPHysics use CLL to search neighbor particles. Although CLL involves more false neighbor particles, this method is a much more general method compared with VL, which has to maintain the information of neighbor particles for each target particle. VL requires much more memory space compared with CLL, but the improvement is very small. Nevertheless, as mentioned above, CLL might result in the problem caused by atomic operations. The returned value of atomic operations is random, which means the index of particles inside each cell is arbitrary. As a result, the particles in a task might distribute in different region space in a cell. For example (see Fig. 3), in the center cell, when task  $t_i$  loads the possible neighbor particles in the left neighbor cells, only the thread of deep blue particle calculates the contributions from possible neighbor particles. The other threads of task  $t_i$  have to wait for the thread of deep blue particle to finish calculating contributions from true neighbor particles. Similarly, when

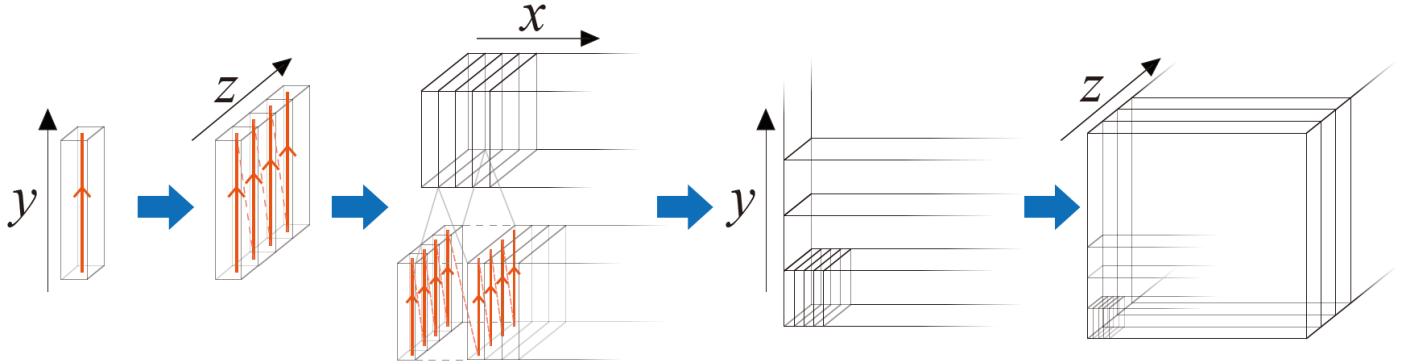


Figure 6: The overview of particle distribution trend in the simulation space. The particle data of every four  $L_4$  cubes remain consecutive along the  $y$ -axis in a  $L_3$  cuboid. The particle data of every four  $L_3$  cuboids remain consecutive along the  $z$ -axis in a  $L_2$  cuboid. The particle data of every  $L_2$  cuboid remain consecutive along the  $x$ -axis.

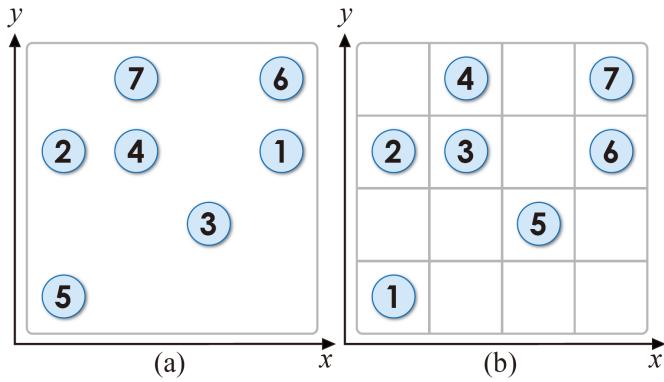


Figure 7: (a) shows the arbitrary distribution of particles in a 2D-cell implemented with traditional uniform grid method. (b) shows the ordered distribution of particles in a 2D-cell implemented with our hierarchical grid method.

<sup>230</sup>  $t_i$  loads the possible neighbor particles in the right neighbor cells, the thread of deep blue particle has to wait for the other threads of task  $t_i$  to finish the calculation. Such a problem can be avoided if we arrange the deep blue particle into task  $t_j$ . So we need to sub-decompose each cell for locating particles in a  
<sup>235</sup> more refined unit space.

In recent years, SPGrid is a popular hierarchical grid, which is associated with Morton encoding [37] [38] and widely used in Material Point Method (MPM) [39] [40]. However, SPGrid is an octree structure grid, so we can not simply combine 27 neighbor cells into 9 cuboids [27]. We need to access each neighbor cell separately, which unavoidably involves more loop iterations. Consequently, we can not take the full advantage of memory consecutiveness, and narrowing neighbor region for each target particle in SPGrid could be quite difficult. Therefore, we propose a novel hierarchical grid method associated with a novel encoding method to improve the coherence of particle distribution. Different from SPGrid, we sub-decompose each unit space of different-level grids into four sub-cuboids. With our organization, we can keep the advantage of uniform grid and improve the coherence of particle distribution. Fig. 4 gives the overview of our hierarchical grid. For the convenience of illustration, we simply decompose simulation space into 27 cells (it is named with  $L_1$  cuboid in this paper) as  $L_1$  grid. In

this grid level, we inherit the uniform grid encoding method to identify each cell. In  $L_2$  grid level, we decompose a cell into 4 cuboids ( $L_2$  cuboid) along the  $x$ -axis, and we decompose a  $L_2$  cuboid into 4 sub-cuboid ( $L_3$  cuboid) along the  $z$ -axis as  $L_3$  grid. In  $L_4$  grid level, we decompose a  $L_3$  cuboid into 4 cubes ( $L_4$  cuboid) along the  $y$ -axis. The indices of  $L_2$ ,  $L_3$  and  $L_4$  cuboid range from 0 to 3, two bits are enough to encode  $L_x$  cuboids. Fig. 5 gives the overview of our encoding method for on our hierarchical grid. For each particle with position  $\mathbf{P} = (x, y, z)$ , we calculate cell ( $L_1$  cuboid) index according to the following formula:

$$I(\mathbf{P}) = \left\lfloor \frac{x}{h} \right\rfloor + \left\lfloor \frac{y}{h} \right\rfloor \cdot X + \left\lfloor \frac{z}{h} \right\rfloor \cdot X \cdot Y, \quad (2)$$

where  $X, Y$  are the numbers of cells of the uniform grid in the  $x$ - and  $y$ -axis. According to our encoding method, we can get the coordinate value by the following formula:

$$I_c(\mathbf{P}_c) = \left\lfloor \frac{y_c}{h_c} \right\rfloor + \left\lfloor \frac{z_c}{h_c} \right\rfloor \cdot 2 + \left\lfloor \frac{x_c}{h_c} \right\rfloor \cdot 4, \quad (3)$$

where  $\mathbf{P}_c = (x_c, y_c, z_c)$  is the relative position inside corresponding cell,  $h_c = h/4$ . Finally the new hash value of particles can be calculated by following formula:

$$H(\mathbf{P}) = I \cdot 64 + I_c. \quad (4)$$

---

#### Algorithm 1 Grid Building and Counting Sort.

---

- 1: //generate grid information
  - 2: calculate hash value of particle using Eq. 2, 3, 6
  - 3:  $\mathbf{C}, \mathbf{f} \leftarrow$  conduct CUDA **atomicAdd** operations on particle hash value
  - 4:  $\mathbf{O} \leftarrow$  conduct GPU-based prefix sum algorithm on  $\mathbf{C}$
  - 5: //sort particles
  - 6: sort particles according to  $\mathbf{C}, \mathbf{O}, \mathbf{f}$
  - 7: rebuild  $\mathbf{C}$  using Eq. 5
- 

After sorting the new hash value of particles, all particles are then mapped to 1D memory and Fig. 6 shows the distribution trend of particles in simulation space. As our new hash value contains the information of higher-level grids, our method can improve the coherence of particle distribution compared

with the traditional uniform grid (see Fig. 7). Our implementation is summarized in Alg. 1, where  $\mathbf{C}$  is counting array of particles for hash value,  $\mathbf{O}$  is offset array of particles for hash value,  $\mathbf{f}$  is the indices array of particles inside cells and  $\mathbb{C}$  is counting array of particles for cells. In order to generate task array, we can rebuild  $\mathbf{C}$  by the following formula:

$$\mathbb{C}(\varepsilon) = \mathbf{O}((\varepsilon + 1) \cdot 64) - \mathbf{O}(\varepsilon \cdot 64), \quad (5)$$

where  $\varepsilon$  is the cell index.

#### 4.2. Hierarchical Neighbor Traversal Strategy

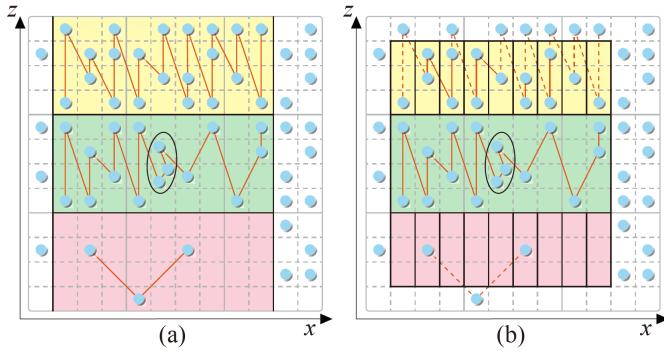


Figure 8: The overview of our hierarchical neighbor traversal strategy. The blue circles represent particle. The ellipses represent tasks. The solid red lines represent two connected particles are consecutive in global memory and both particles should be loaded for the task. The regions colored with yellow, green and blue represent neighbor regions.

In the process of neighbor traversal, Fig. 3 shows that there are several false neighbor particles in neighbor cells. In order to decrease the access of false particles, some researchers propose VL based methods, which records all true neighbor particles for each particle. Obviously, these VL based methods require much more memory space compared with CLL based methods, but the improved performance is quite limited. [29] proposed a simple sub-decomposing method to construct VL. Their method also can be applied to CLL, but without a good organization of grid, this method would cause high overhead of loop iterations.

The main purpose of our hierarchical grid method is to efficiently reduce the neighbor space for target particles. Based on our grid method, we propose a novel neighbor traversal strategy, which dynamically loads neighbor particles in different level grids according to the distribution and the continuity of particles. Our method can greatly decrease the access of false neighbor particles with relatively low overhead of loop iterations. For the convenience of illustration, we demonstrate our hierarchical neighbor traversal strategy in 2D simulation space. In the process of generating tasks, we calculate the range of  $L_2$  cuboid index for each task. In Fig. 8(a), all particles of the target task (circled with ellipse) are in the second  $L_2$  cuboid of the center cell. According to the range of  $L_2$  cuboid index, we can narrow the range of neighbor region in the  $x$ -axis. As the particles are consecutive along the  $x$ -axis, we just need to calculate the offset value of particles in the most left  $L_2$  cuboid and the

number of particles inside the consecutive region colored with same color (it is named with cuboid neighbor cell). In this grid level, we avoid the loop iterations in higher-level grid and decrease the access of some false neighbor particles. Alg. 2 has outlined our method of narrowing the search space in  $L_2$ -level grid.  $\mathbf{f}_s$  represents the index of the cuboid neighbor cell, which is being accessed by threads.  $\mathbf{u}_s$  represents the count of neighbor particles, which have been accessed in cuboid cell  $\mathbf{f}_s$ .  $\mathbf{l}_s$  represents the index of  $L_2$  cuboid, which is being accessed by threads.

---

#### Algorithm 2 Process of Building Cuboid Cells.

---

```

1: for all threads of a CTA do
2:    $\text{Cell}_{\text{pos}} \leftarrow$  get cell position from target task
3:   //s is the thread index in CTA
4:   if  $s < 9$  then
5:     locate each thread to cuboid neighbor cell  $t$ 
6:     // $x_{\min}$  is the index of the most left  $L_2$  cuboid
7:     // $x_{\max}$  is the index of the most right  $L_2$  cuboid
8:      $\mathbf{A}_s \leftarrow$  get particle offset according to  $x_{\min}$  and  $x_{\max}$ 
9:      $\mathbf{B}_s \leftarrow$  get particle count according to  $x_{\min}$  and  $x_{\max}$ 
10:    end if
11:    if  $s == 31$  then
12:       $\text{SC} \leftarrow$  get the number of  $L_2$  cuboids in cuboid cell
13:    end if
14:    syncthreads
15:     $\mathbf{f}_s \leftarrow 0$ ,  $\mathbf{u}_s \leftarrow 0$ ,  $\mathbf{l}_s \leftarrow 0$ 
16:  end for

```

---

In our hierarchical grid, we also can search neighbor particles in  $L_3$  cuboid. The range of  $L_3$  cuboid index of the task is from 2 to 3 (see Fig. 8(b)). Similarly, according to the range of  $L_3$  cuboid index, we can further reduce the range of neighbor region in the  $z$ -axis. In  $L_3$  grid level, reduce search space in the  $z$ -axis might break the continuity of neighbor particles, which means we can not load neighbor particles in a consecutive memory domain. Instead, we need to load the particles in each  $L_2$  sub-cuboid iteratively, which might result in the unnecessary waste of computational resources. For example, in Fig. 8(b), there are only two neighbor particles in the cuboid neighbor cell colored with pink. In this cuboid neighbor cell, if we search neighbor particles in each  $L_2$  sub-cuboid iteratively, we only decrease the access of one false neighbor particle, while a lot of invalid loop iterations are produced. Therefore, before reduce the search space in the  $z$ -axis, we need to consider the distribution of particles in the corresponding neighbor space. In order to distinguish different particle distributions in different neighbor space, we define a threshold  $T$ . Through the comparison of particle count in each cuboid neighbor cell and  $T$ , we can determine whether to load neighbor particles in each  $L_2$  sub-cuboid iteratively or directly load neighbor particles in the corresponding cuboid neighbor cell. With this dynamic strategy, we can reduce the unnecessary overhead of loop iterations. In addition, with the help of shared memory and threads cooperation, our method can greatly improve the performance of GpuSPHCTA. Alg. 3 has summarized our neighbor traversal implementation, where GMEM is global memory, SMEM is shared memory.

Although we can also reduce the range of neighbor space in the  $y$ -axis, such operation will unavoidably result in much more overhead of loop iterations, which greatly declines the performance. Through our experiments, we find that it is sufficient to reduce the range of neighbor space in the  $x$ - and  $z$ -axis. The main purpose of  $L_4$  grid is to improve particle coherence.

### Algorithm 3 Loading Strategy of Neighbor Particles.

```

1: for all threads of a CTA do
2:   while true do
3:     if  $\mathbf{u}_s == \mathbf{B}[\mathbf{f}_s]$  or  $\mathbf{l}_s == \text{SC}$  then
4:        $\mathbf{f}_s \leftarrow \mathbf{f}_s + 1$ ,  $\mathbf{u}_s \leftarrow 0$ ,  $\mathbf{l}_s \leftarrow 0$ 
5:       if  $9 \leq \mathbf{f}_d$  then
6:         break
7:       end if
8:     end if
9:     if cuboid cell  $\mathbf{f}_s$  is cut in the  $z$ -axis &&  $\mathbf{B}[\mathbf{f}_s] > \mathbb{T}$  then
10:       $\mathbf{a}_s \leftarrow$  get particle offset of  $L_2$  sub-cuboid according
11:        to  $z_{min}, z_{max}, \mathbf{A}[\mathbf{f}_s]$  and  $\mathbf{l}[\mathbf{f}_s]$ 
12:       $\mathbf{b}_s \leftarrow$  get particle count of  $L_2$  sub-cuboid according
13:        to  $z_{min}, z_{max}, \mathbf{A}[\mathbf{f}_s]$  and  $\mathbf{l}[\mathbf{f}_s]$ 
14:      if  $\mathbf{u}_s == \mathbf{b}_s$  then
15:         $\mathbf{l}_s \leftarrow \mathbf{l}_s + 1$ ,  $\mathbf{u}_s \leftarrow 0$ 
16:        if  $\mathbf{l}_s == \text{SC}$  then
17:          continue
18:        end if
19:      end if
20:      if  $s < \min(32, \mathbf{b}[\mathbf{f}_s] - \mathbf{u}_s)$  then
21:         $pi \leftarrow \mathbf{a}_s + \mathbf{u}_s + s$ 
22:        read particle  $pi$  from GMEM to SMEM
23:      end if
24:       $\mathbf{u}_s \leftarrow \mathbf{u}_s + \min(32, \mathbf{b}_s - \mathbf{u}_s)$ 
25:    else
26:      if  $s < \min(32, \mathbf{B}[\mathbf{f}_s] - \mathbf{u}_s)$  then
27:         $pi \leftarrow \mathbf{A}[\mathbf{f}_s] + \mathbf{u}_s + s$ 
28:        read particle  $pi$  from GMEM to SMEM
29:      end if
30:       $\mathbf{u}_s \leftarrow \mathbf{u}_s + \min(32, \mathbf{B}[\mathbf{f}_s] - \mathbf{u}_s)$ 
31:    end if
32:  end while
33: end for

```

### 4.3. Hierarchical Task Schedule Strategy

With the help of hierarchical perpendicular grid, the coherence of particle distribution is improved. Consequently, the locality of consecutive tasks is improved. Therefore, two consecutive tasks in the same cell always share a great part of neighbor particles (we name these tasks as Task Pair). In GpuSPHCTA, each CTA deals with one task separately, the overlap of neighbor particles is overloaded for Task Pair, which is an unnecessary waste of computational resources. So we proposed a hierarchical task assignment strategy by merging two consecutive tasks in the same cell into a bigger task. There are two kinds

of tasks in our framework (we name these two kinds of tasks as  $K_1$  task and  $K_2$  task). These two kinds of tasks are launched simultaneously without any additional time cost of CUDA kernel function launching. And we will detail our strategy here.

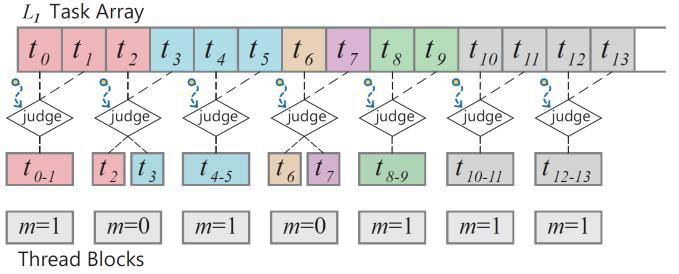


Figure 9: The overview of our tasks merging strategy. The blocks colored with the same color represent these tasks belong to the same cell.

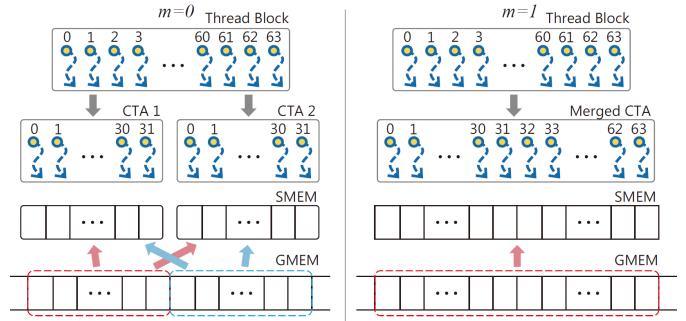


Figure 10: The overview of different CTAs' cooperation strategy for different value of  $m$ .

As shown in Fig. 9, the particles of each cell are divided into several  $K_1$  tasks. In order to find the  $K_1$  tasks, which can be merged as  $K_2$  tasks. We assign a thread for each task pair to determine whether these two  $K_1$  tasks can be merged. If such two  $K_1$  tasks belong to the same cell, we merge the two  $K_1$  tasks into a  $K_2$  task. In our implementation, we propose a signal value  $m$  to denote tasks, if  $m = 1$ ,  $K_1$  task has been merged; if  $m = 0$ ,  $K_1$  task has not been merged.

As the size of tasks might be different in our hierarchical task strategy, we should launch CUDA thread blocks with different thread sizes for catering to different tasks. However, in CUDA model, we can only launch CUDA thread blocks with same thread size. we thus use threads merging strategy to construct different thread groups with different thread sizes. As we have recorded the signal value  $m$  for each task to determine whether the task should be merged, we can easily identify different size tasks in different thread blocks. In GpuSPHCTA, the thread index of each CTA is calculated according to  $s = \tau \% 32$ , and the CTA index is calculated according to  $w = \lfloor \tau / 32 \rfloor$ . In order to identify different size tasks, we use a new thread index calculation function in the following form:

$$s = \tau \% 32 + m \cdot w \cdot 32, \quad (6)$$

where  $\tau$  is the thread index of CUDA thread block.

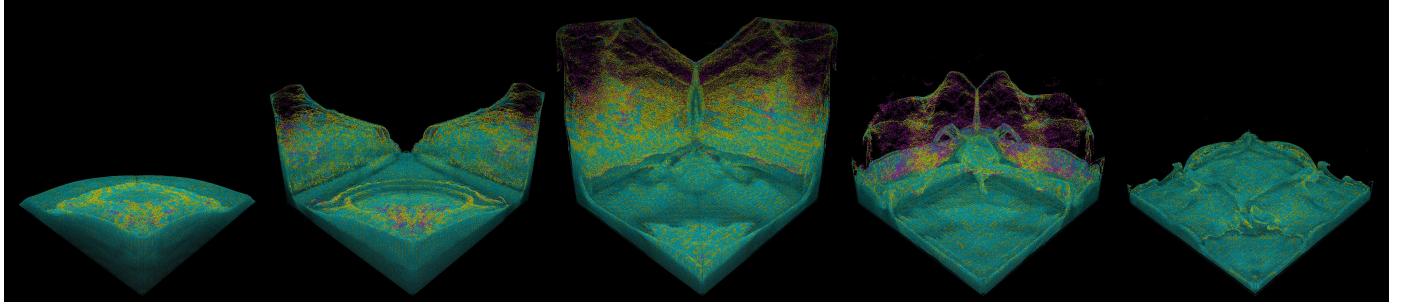


Figure 11: The scene of dam break shows the hierarchical task structure in our framework. The green particles are arranged into  $K_1$  tasks; The yellow particles are arranged into  $K_2$  tasks; the red particles are dealt with traditional method (DualSPHysics). In our experiments, our framework can exhibit 1.73x speedup compared with GpuSPHCTA without any compromise of numerical accuracy and sacrifice of simulation detail.

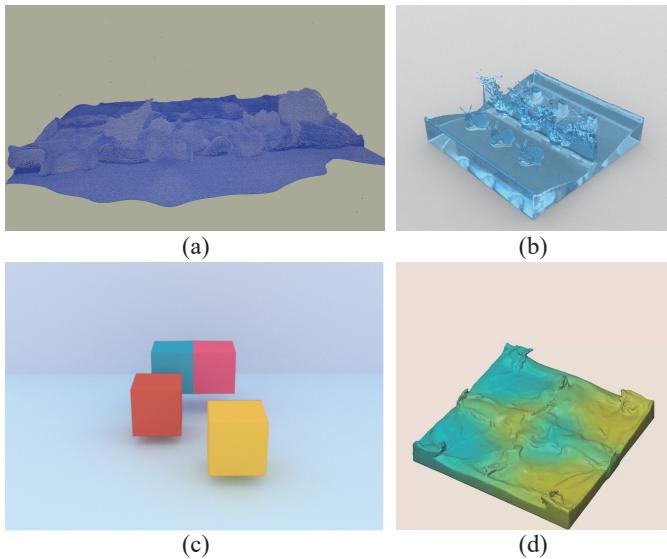


Figure 12: Four benchmarks for evaluating the performance of our framework tested with four different SPH algorithms. benchmark (a) is ocean wave. benchmark (b) is the dropping of bunny rabbits. benchmark (c) is the interaction of dropping elastic cubes. benchmark (d) is the interaction of different miscible fluids.

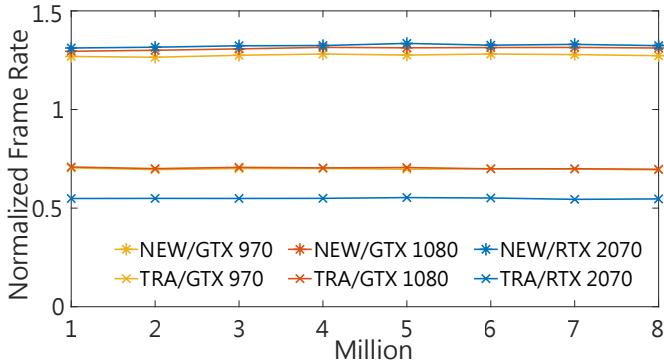


Figure 13: The performance results of different frameworks tested with different particle number on different hardware platforms. The frame rate is normalized to the performance of GpuSPHCTA.

Fig. 10 shows the differences of thread blocks with different value of  $m$ . If  $m = 0$ , two CTAs deal with corresponding  $K_1$  tasks separately; if  $m = 1$ , two CTAs are merged as 64 threads group; threads in a new thread group working together to load neighbor particles from global memory to shared memory (when two tasks in a thread block are denoted to be merged, the particle distribution range of new task might be changed, so we need to update the maximum and minimum value of  $L_2$  cuboid index and  $L_3$  cuboid index according to the corresponding values of merged tasks). Each target particle in the corresponding thread block loads all these neighbor particles to calculate attributes. Therefore, our merged task assignment strategy can avoid loading overlap of neighbor particles. Moreover, our task assignment strategy can also reduce the overhead of loop iterations, because the new thread group can load twice the number of neighbor particles in each iteration.  $K_1$  task strategy can reduce the idle threads, and  $K_2$  task strategy can reduce the unnecessary loading of particles. So our hierarchical task strategy can well cater to the heterogeneous particle distribution in different cells. The key components of our framework are summarized in Alg. 4.

---

#### Algorithm 4 The Overview of Our Framework.

---

```

1: repeat
2:   //preprocessing
3:   conduct preprocessing according to Alg. 1
4:   merge task pair according to the process of Fig. 9
5:   update the range of  $L_2$  and  $L_3$  cuboid index for  $K_2$  tasks
6:   //each attributes calculation step
7:   generate cuboid cells according to Alg. 2
8:   load neighboring particle data according to Alg. 3
9:   calculate attributes for target particles
10:  until the end of simulation

```

---

## 5. Experimental Results and Evaluations

In this section, we use six benchmarks implemented with different SPH algorithms to evaluate our framework (NEW). The comparison objects are DualSPHysics (TRA) and GpuS-

Table 1: Test case settings and simulation results. The results are the average of the first 640 time steps. **Pre** is preprocessing overhead. **Dfor** is the overhead for force computation. **Tot** is total simulation overhead.

GPU		#particles	AC	DualSPHysics (ms)			GpuSPHCTA (ms)			OUR METHOD(ms)		
				Pre	Dfor	Tot	Pre	Dfor	Tot	Pre	Dfor	Tot
<b>GTX 970</b>	case 1	8,000,000	64	8.9	510.3	519.3	13.5	331.4	344.9	15.0	326.3	341.3
	case 2		125	9.2	990.7	999.9	14.1	665.1	679.2	15.0	576.4	591.4
	case 3		297	10.0	2315.2	2325.2	15.6	1626.0	1641.6	15.7	1264.4	1280.1
	case 1		64	8.2	227.7	235.9	10.8	147.2	158	11.6	139.7	151.3
	case 2		125	7.9	439.9	447.8	11.1	293.6	304.7	11.8	246.1	257.9
	case 3		297	8.3	1015.6	1023.9	12.0	705.3	717.3	12.3	532.5	544.8
	case 1		64	15.9	247.6	263.5	18.5	122.4	140.9	16.6	121.0	137.6
	case 2		125	15.8	501.7	517.5	18.8	262.0	280.8	17.6	220.1	237.7
	case 3		297	18.3	1171.9	1190.2	21.2	632.8	654.0	17.2	475.0	492.2

PHCTA. The experiments are performed on different hardware platforms as follows:

- Intel(R) Core(TM) i7 9700K and NVIDIA Geforce GTX 970 (Maxwell Architecture).
- Intel(R) Core(TM) i7 9700K and NVIDIA Geforce GTX 1080 (Pascal Architecture).
- Intel(R) Core(TM) i7 9750H and NVIDIA Geforce RTX 2070 (Turing Architecture).

### 5.1. Parameter Settings and Effects

In order to find the correlation between particle number and speedup rate of our framework compared with GpuSPHCTA and DualSPHysics, the benchmark we used here is cuboid with constant size, which means we omit the process of updating the position of particles for ensuring that the average particle number of nonempty cells (it is named with  $AC$ ) keeps constant in each frame. The number of particles ranges from one million to eight million, and  $AC$  is set to 297. Fig. 13 shows the results of our experiments, which indicate that the speedup rate of our framework has no obvious correlation with particle number.

In addition, we use the same benchmark to reveal the correlation between  $AC$  and the speedup rate of our framework. Table 1 gives the key information of the test cases and detailed elapsed time. The results of our test cases indicate that the speedup rate of our framework compared with GpuSPHCTA and DualSPHysics is growing with  $AC$ . Both Table 1 and Fig. 13 show that shared memory based method (GpuSPHCTA and our method) can exhibit better speedup on RTX 2070 compared with DualSPHysics (our method can exhibit 2.38x speedup on RTX 2070 tested with case 3). In addition, we find that the traditional method (DualSPHysics) can exhibit better performance on GTX 1080.

Overall, the speedup rate of our framework is closely related to  $AC$ , regardless of particle number. In our test cases, the performance of our method is the best while the traditional method is relatively slow.

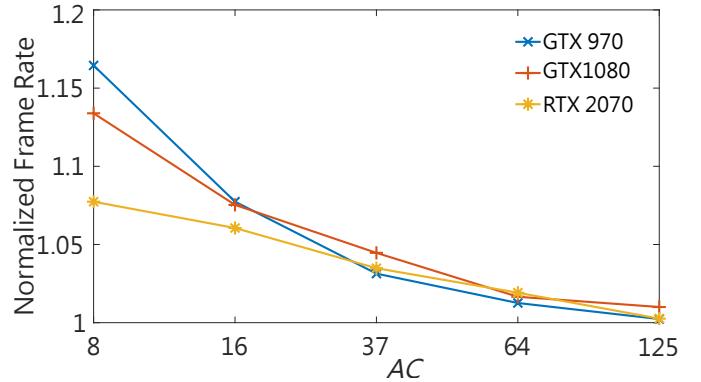


Figure 14: The performance results of our grid method. The frame rate is normalized to the performance of SPGrid.

### 5.2. Improved Performance

The experiments mentioned above reveal that  $AC$  is the key parameter. However, the experiments on cuboid with constant size can not reveal real performance improvement of our method.  $AC$  might be different in each frame, which is the result of the advection of particles. Thus we need to test our framework with real simulations, and the benchmark we used here is dam break (see Fig. 11).

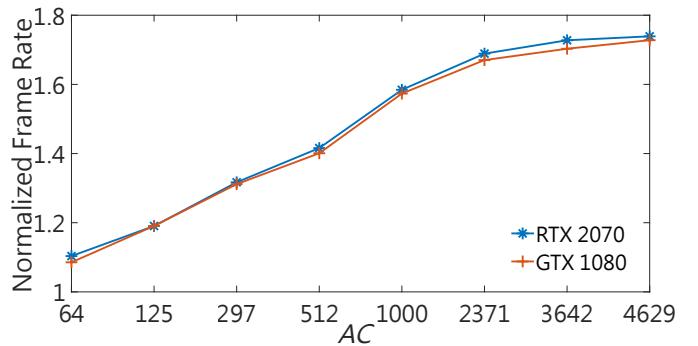


Figure 15: The performance results of dam break tested with different  $AC$ . The frame rate is normalized to the performance of GpuSPHCTA.

First, we compare our grid method with SPGrid [39] on different GPUs. Fig. 14 shows the results of our experiments, which indicate that our grid method can obviously exhibit performance improvement if  $AC$  is relatively small. When  $AC$  exceeds 125, the performance of our grid method and SPGrid is about the same. Our grid method can reduce loop iterations, but the time saved from loop iterations accounts for a smaller part of the total time when  $AC$  is increased. Both SPGrid and our grid can improve the coherence of particle distribution, but our grid method can make it much easier to reduce neighbor space.

Second, we compare our framework with GpuSPHCTA (the performance of GpuSPHCTA is better than DualSPHysics in our experiments). As the efficiency of GTX 970 is the slowest in our three GPUs, we will not conduct our next experiments on GTX 970. Fig. 15 also shows that the speedup rate of our framework is growing with an initial  $AC$ . As a simulation with initial  $AC$  exceeding 5000 is rare, we have not tested our framework with bigger initial  $AC$ . In our experiments, our framework can exhibit 1.73x speedup.

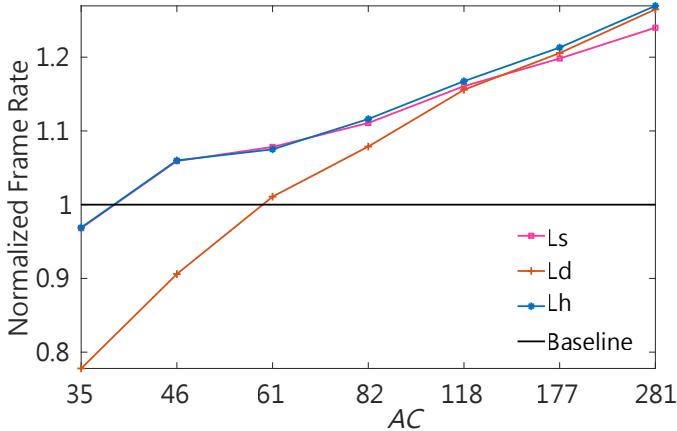


Figure 16: The performance results of neighbor traversal strategies. The frame rate is normalized to the performance of GpuSPHCTA.

Finally, we test the performance of our dynamic strategy for reducing neighbor space on RTX 2070. Fig. 16 shows the results of our experiments. Ls represents the method which only reduce neighbor space by  $L_2$  cuboid index (see Fig. 8(a)); Ld represents the method which straightforwardly reduce neighbor space by  $L_2$  and  $L_3$  cuboid index (see Fig. 8(b)); Lh represents our dynamic strategy. In the experiments whose  $AC$  is smaller than 120, the performance of Ls is better than Ld, which is caused by more loop iterations in Ld. In the experiments whose  $AC$  is bigger than 120, the performance of Ld is better than Ls, because the time cost of additional loop iteration in Ld is smaller than the time saved by avoiding loading some false neighbor particles. As Lh possesses both the advantage of Ls and Ld, Lh performs well in all these experiments ( $T$  is set to 90). Moreover, we find that GpuSPHCTA has better performance when initial  $AC$  is smaller than 40, which is the result of the more overhead of preprocessing in our framework when  $AC$  is small.

### 5.3. Simulation Evaluations

To some extent, dam break is a simple physical simulation, and there are several variants of SPH for some relatively complex simulations. In order to reveal the generality and availability of our framework, we test our framework with the other four benchmarks implemented with four different SPH algorithms.

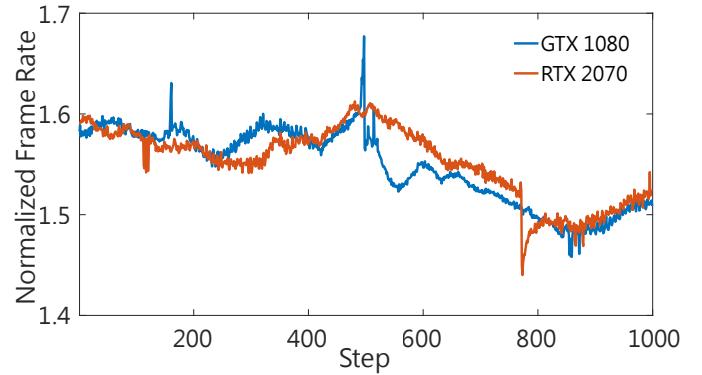


Figure 17: The performance results of ocean wave simulation. The frame rate is normalized to the performance of GpuSPHCTA.

The first benchmark is ocean wave implemented with WC-SPH [41] (see Fig. 12(a)). The initial  $AC$  of this benchmark is about 1000, and the number of simulation particles is 17,970,160. Fig. 17 gives the test results of our framework compared with GpuSPHCTA on different hardware platforms. It can be seen from Fig. 17 that the speedup rate keeps going up and going down, because the particles keep splashing and converging, which means  $AC$  keeps going up and going down. Our framework can exhibit about 1.55x speedup.

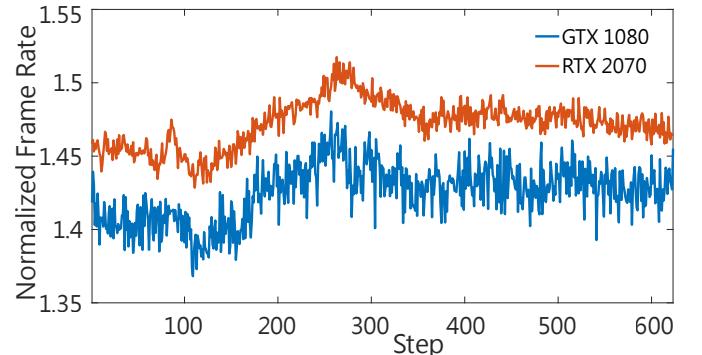


Figure 18: The performance results of dropping bunny rabbits simulation. The frame rate is normalized to the performance of GpuSPHCTA.

The second benchmark is the dropping of bunny implemented with PCISPH [42] (see Fig. 12(b)). PCISPH involves much computation process for great incompressibility. The initial  $AC$  of this benchmark is about 500, and the number of simulation particles is 1,516,563. The test results (see Fig. 18) indicate that the change of speedup rate is relatively small, which is the result of the great incompressibility of PCISPH. Furthermore, Fig. 18 shows that our framework can exhibit better speedup on RTX 2070 compared with GTX 1080; our framework can

exhibit about 1.43x speedup on GTX 1080, 1.47x speedup on RTX 2070.

The third benchmark is the mixture of two different fluid implemented with multiple fluid SPH [18] (see Fig. 12(d)). This SPH algorithm needs to calculate the attributes of different fluids, so the simulation of this benchmark is complex. The initial AC of this benchmark is about 500, and the number of simulation particles is 3,169,044. Similarly, the speedup rate changes with AC, and finally keeps stable as the fluid almost keeps constant (see Fig. 19). our framework can exhibit about 1.47x speedup.

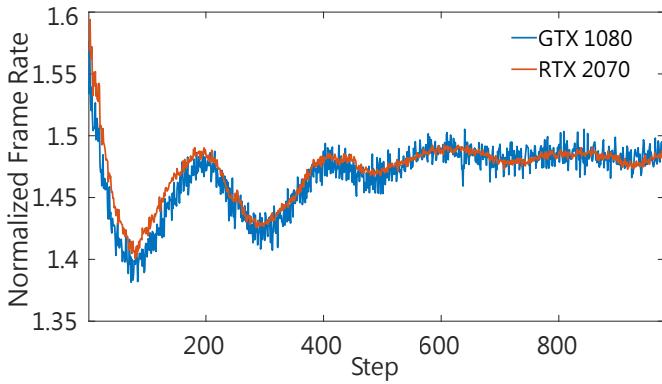


Figure 19: The performance results of multiple fluid simulation. The frame rate is normalized to the performance of GpuSPHCTA.

The fourth benchmark is elastic solid implemented with multiple phase SPH [19] (see Fig. 12(c)), which is the most complex simulation algorithm in our experiments. The initial AC of this benchmark is about 500, and the number of simulation particles is 2,788,192. Fig. 20 shows the performance of our framework. There is no splash of particles in the simulation of elastic solid and the deformation of elastic solid is relatively small, which means the change of AC is very small. Therefore, the relative performance of our framework almost keeps stable in this simulation. Our framework can exhibit about 1.61x speedup on GTX 1080, 1.63x speedup on RTX 2070.

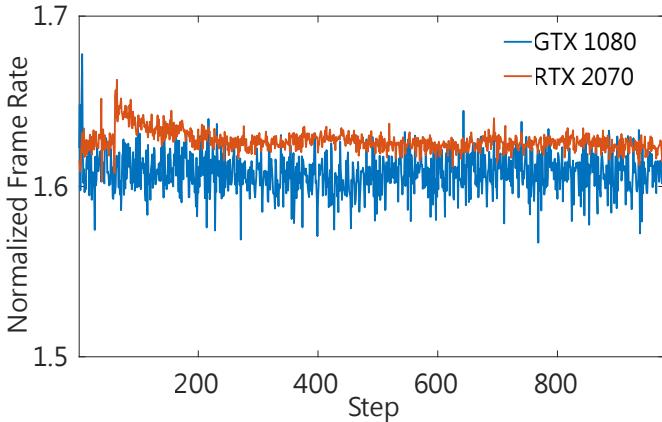


Figure 20: The performance results of elastic solids simulation. The frame rate is normalized to the performance of GpuSPHCTA.

## 6. Conclusion and Future Works

This paper has detailed an efficient framework with well designed hierarchical task assignment and neighbor search strategies in a new hierarchical grid. The framework takes the full advantage of GPGPU and can greatly reduce the cost of computational resources, so as to further break the performance bottleneck of SPH algorithms based on GpuSPHCTA. The framework has exhibited up to 1.92x speedup without any compromise of numerical accuracy and sacrifice of simulation detail. Therefore, our method can be applied to most of the existing SPH algorithms directly. Even though the framework could not exhibit magnificent speedup compared with GpuSPHCTA, the current solution is both efficient and of general purpose.

As shown in Fig. 5, we shift the binary code of traditional hash value (cell index in this paper) to the left by 6 bits, which means that the counting array of traditional uniform grid used for counting in the sorting algorithm is expanded by 64 times. As a result, the sorting cost is unavoidably increased. In addition, we have to maintain the information of spatial range for computational tasks, so our new strategies need additional preprocess routines. Thus, the preprocessing cost is more than that in GpuSPHCTA, which means our framework might not be suitable for simulations with very few and sparse particles, whose preprocessing cost occupies a significant portion of simulation time. Through our tests, we observe that the increased overhead mainly results from the sorting process. Hence we need to design a much more efficient GPU based sorting algorithm to further improve the performance of our framework. Moreover, as shown in Fig. 15 and Fig. 16, compared with GpuSPHCTA, the improvement of our framework is quite significant when AC is relatively big. On the other hand, GpuSPHCTA can exhibit better performance than our method when AC is very small. Therefore, we could merge GpuSPHCTA into our framework to conduct the simulations whose AC is very small for avoiding the performance degradation.

Although we define a threshold  $\mathbb{T}$  to identify different neighbor traversal strategies in different neighbor regions, constant  $\mathbb{T}$  might not be best suitable to the heterogeneous distribution of particles well, so we need a better strategy to dynamically change the value of  $\mathbb{T}$  for catering to the heterogeneous distribution of particles. Besides, we wish to design a much more general framework, which not only has a better fit for SPH algorithms but also could be applied to other particle systems such as Material Point Method (MPM) in our future work. We also expect to further utilize the concurrency of CTA to design a novel particle merging strategy towards the further improved implementation of a novel adaptive SPH based on GPGPU.

## Acknowledgements

This paper is partially supported by Natural Science Foundation of China under Grants 61532002 and 61672237, National Science Foundation of USA (IIS-1715985 and IIS-1812606).

## References

- [1] S. Green, Cuda particles, NVIDIA Whitepaper 2 (3.2) (2008) 1.

- [2] A. Héroult, G. Bilotta, R. A. Dalrymple, Sph on gpu with cuda, *Journal of Hydraulic Research* 48 (S1) (2010) 74–79.
- [3] J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, J. C. Marongiu, Neighbour lists in smoothed particle hydrodynamics, *International Journal for Numerical Methods in Fluids* 67 (12) (2011) 2026–2042.
- [4] D. Winkler, M. Rezavand, W. Rauch, Neighbour lists for smoothed particle hydrodynamics on gpus, *Computer Physics Communications* 225 (2018) 140–148.
- [5] A. J. C. Crespo, J. M. Domínguez, B. D. Rogers, M. Gómez-Gesteira, S. M. Longshaw, R. B. Canelas, R. Vacondio, A. Barreiro, O. García-Feal, Dualspysics: Open-source parallel CFD solver based on smoothed particle hydrodynamics (SPH), *Computer Physics Communications* 187 (2015) 204–216.
- [6] D. Winkler, M. Meister, M. Rezavand, W. Rauch, gpuphase - A shared memory caching implementation for 2d SPH using CUDA, *Computer Physics Communications* 213 (2017) 165–180.
- [7] D. Winkler, M. Rezavand, M. Meister, W. Rauch, gpuphase - A shared memory caching implementation for 2d SPH using CUDA (new version announcement), *Computer Physics Communications* 235 (2019) 514–516.
- [8] P. Goswami, P. Schlegel, B. Solenthaler, R. Pajarola, Interactive SPH simulation and rendering on the GPU, in: *Proceedings of the 2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, SCA 2010, Madrid, Spain, 2010, Eurographics Association, 2010, pp. 55–64.
- [9] K. Huang, J. Ruan, Z. Zhao, C. Li, C. Wang, H. Qin, A general novel parallel framework for sph-centric algorithms, *Proc. ACM Comput. Graph. Interact. Tech.* 2 (1) (2019) 7:1–7:16.
- [10] R. A. Gingold, J. J. Monaghan, Smoothed particle hydrodynamics: theory and application to non-spherical stars, *Monthly Notices of the Royal Astronomical Society* 181 (3) (1977) 375–389.
- [11] J. J. Monaghan, Smoothed particle hydrodynamics, *Annual Review of Astronomy and Astrophysics* 30 (1) (1992) 543–574.
- [12] M. Müller, D. Charypar, M. H. Gross, Particle-based fluid simulation for interactive applications, in: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, San Diego, CA, USA, July 26–27, 2003, 2003, pp. 154–159.
- [13] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, M. Teschner, SPH fluids in computer graphics, in: *Eurographics 2014 - State of the Art Reports*, Strasbourg, France, April 7–11, 2014, 2014, pp. 21–42.
- [14] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. H. Gross, M. Alexa, Point based animation of elastic, plastic and melting objects, in: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Grenoble, France, August 27–29, 2004, 2004, pp. 141–151.
- [15] D. Gerszewski, H. Bhattacharya, A. W. Bargteil, A point-based method for animating elastoplastic solids, in: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA 2009, New Orleans, Louisiana, USA, August 1–2, 2009, 2009, pp. 133–138.
- [16] R. Keiser, B. Adams, D. Gasser, P. Bazzi, P. Dutré, M. H. Gross, A unified lagrangian approach to solid-fluid animation, in: *Symposium on Point Based Graphics*, Stony Brook, NY, USA, 2005. Proceedings, 2005, pp. 125–133.
- [17] N. Akinci, J. Cornelis, G. Akinci, M. Teschner, Coupling elastic solids with smoothed particle hydrodynamics fluids, *Journal of Visualization and Computer Animation* 24 (3–4) (2013) 195–203.
- [18] B. Ren, C. Li, X. Yan, M. C. Lin, J. Bonet, S. Hu, Multiple-fluid SPH simulation using a mixture model, *ACM Transactions on Graphics* 33 (5) (2014) 171:1–171:11.
- [19] X. Yan, Y. Jiang, C. Li, R. R. Martin, S. Hu, Multiphase SPH simulation for interactive fluids and solids, *ACM Transactions on Graphics* 35 (4) (2016) 79:1–79:11.
- [20] T. Yang, J. Chang, M. C. Lin, R. R. Martin, J. J. Zhang, S. Hu, A unified particle system framework for multi-phase, multi-material visual simulations, *ACM Transactions on Graphics* 36 (6) (2017) 224:1–224:13.
- [21] J. Bender, D. Koschier, Divergence-free SPH for incompressible and viscous fluids, *IEEE Trans. Vis. Comput. Graph.* 23 (3) (2017) 1193–1206.
- [22] D. Koschier, J. Bender, B. Solenthaler, M. Teschner, Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids, in: W. Jakob, E. Puppo (Eds.), *Eurographics 2019 - Tutorials*, The Eurographics Association, 2019.
- [23] P. Kipfer, M. Segal, R. Westermann, Overflow: a gpu-based particle engine, in: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004*, Grenoble, France, August 29–30, 2004, 2004, pp. 115–122.
- [24] A. Kolb, L. Latta, C. Rezk-Salama, Hardware-based simulation and collision detection for large particle systems, in: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004*, Grenoble, France, August 29–30, 2004, 2004, pp. 123–131.
- [25] A. Kolb, N. Cuntz, Dynamic particle coupling for gpubased fluid simulation, in: *Proceedings of 18th Symposium on Simulation Technique*, 2005, pp. 722–727.
- [26] T. Harada, S. Koshizuka, Y. Kawaguchi, Smoothed particle hydrodynamics on gpus, in: *Proceedings of computer graphic*, 2007, pp. 63–70.
- [27] J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method, *Computer Physics Communications* 184 (3) (2013) 617–627.
- [28] X. Xia, Q. Liang, A gpu-accelerated smoothed particle hydrodynamics (SPH) model for the shallow water equations, *Environmental Modelling and Software* 75 (2016) 28–43.
- [29] K. Ohno, T. Nitta, H. Nakai, Sph-based fluid simulation on GPU using verlet list and subdivided cell-linked list, in: *Fifth International Symposium on Computing and Networking*, CANDAR 2017, Aomori, Japan, November 19–22, 2017, 2017, pp. 132–138.
- [30] F. Zhang, L. Hu, J. Wu, X. Shen, A sph-based method for interactive fluids simulation on the multi-gpu, in: *Proceedings of the 10th International Conference on Virtual Reality Continuum and its Applications in Industry*, VRCAI 2011, Hong Kong, China, December 11–12, 2011, 2011, pp. 423–426.
- [31] L. Hu, X. Shen, X. Long, Research on SPH parallel acceleration strategies for multi-gpu platform, in: *Advanced Parallel Processing Technologies - 10th International Symposium*, APPT 2013, Stockholm, Sweden, August 27–28, 2013, Revised Selected Papers, 2013, pp. 104–118.
- [32] E. Rustico, G. Bilotta, G. Gallo, A. Héroult, C. D. Negro, Smoothed particle hydrodynamics simulations on multi-gpu systems, in: *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP 2012, Munich, Germany, February 15–17, 2012, 2012, pp. 384–391.
- [33] E. Rustico, G. Bilotta, A. Héroult, C. D. Negro, G. Gallo, Advances in multi-gpu smoothed particle hydrodynamics simulations, *IEEE Transactions on Parallel and Distributed Systems* 25 (1) (2014) 43–52.
- [34] D. Valdez-Balderas, J. M. Domínguez, B. D. Rogers, A. J. C. Crespo, Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-gpu clusters, *Journal of Parallel and Distributed Computing* 73 (11) (2013) 1483–1493.
- [35] J. M. Domínguez, A. J. C. Crespo, D. Valdez-Balderas, B. D. Rogers, M. Gómez-Gesteira, New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters, *Computer Physics Communications* 184 (8) (2013) 1848–1860.
- [36] K. Verma, K. Szewc, R. Wille, Advanced load balancing for SPH simulations on multi-gpu architectures, in: *2017 IEEE High Performance Extreme Computing Conference*, HPEC 2017, Waltham, MA, USA, September 12–14, 2017, 2017, pp. 1–7.
- [37] G. Morton, A computer oriented geodetic data base and a new technique in file sequencing (01 1966).
- [38] D. Holzmüller, Efficient neighbor-finding on space-filling curves, *CoRR* abs/1710.06384 (2017). [arXiv:1710.06384](https://arxiv.org/abs/1710.06384).
- [39] R. Setaluri, M. Aanjaneya, S. Bauer, E. Sifakis, Spgrid: a sparse paged grid structure applied to adaptive smoke simulation, *ACM Trans. Graph.* 33 (6) (2014) 205:1–205:12.
- [40] M. Gao, X. Wang, K. Wu, A. Pradhana, E. Sifakis, C. Yuksel, C. Jiang, GPU optimization of material point methods, *ACM Trans. Graph.* 37 (6) (2018) 254:1–254:12.
- [41] M. Becker, M. Teschner, Weakly compressible SPH for free surface flows, in: M. Gleicher, D. Thalmann (Eds.), *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA 2007, San Diego, California, USA, August 2–4, 2007, Eurographics Association, 2007, pp. 209–217.
- [42] B. Solenthaler, R. Pajarola, Predictive-corrective incompressible SPH, *ACM Trans. Graph.* 28 (3) (2009) 40.