A General Novel Parallel Framework for SPH-centric Algorithms

KEMENG HUANG, East China Normal University, China
JIMING RUAN, East China Normal University, China
ZIPENG ZHAO, East China Normal University, China
CHEN LI, East China Normal University, China
CHANGBO WANG*, East China Normal University, China
HONG QIN, Stony Brook University, United States

To date, large-scale fluid simulation with more details employing the Smooth Particle Hydrodynamics (SPH) method or its variants is ubiquitous in computer graphics and digital entertainment applications. Higher accuracy and faster speed are two key criteria evaluating possible improvement of the underlying algorithms within any available framework. Such requirements give rise to high-fidelity simulation with more particles and higher particle density that will unavoidably increase computational cost significantly. In this paper, we develop a new general GPGPU acceleration framework for SPH-centric simulations founded upon a novel neighbor traversal algorithm. Our novel parallel framework integrates several advanced characteristics of GPGPU architecture (e.g., shared memory and register memory). Additionally, we have designed a reasonable task assignment strategy, which makes sure that all the threads from the same CTA belong to the same cell of the grid. With this organization, big bunches of continuous neighboring data can be loaded to the shared memory of a CTA and used by all its threads. Our method has thus low global-memory bandwidth consumption. We have integrated our method into both WCSPH and PCISPH, that are two improved variants in recent years, and demonstrated its performance with several scenarios involving multiple-fluid interaction, dam break, and elastic solid. Through comprehensive tests validated in practice, our work can exhibit up to 2.18× speedup when compared with other state-of-the-art parallel frameworks.

CCS Concepts: • **Computing methodologies → Physical simulation**; **Massively parallel algorithms**; **Shared memory algorithms**.

Additional Key Words and Phrases: smooth particle hydrodynamics (SPH), cooperative thread array (CTA), shared memory, GPGPU architecture, fluid simulation, solid simulation

ACM Reference Format:
Kemeng Huang, Jiming Ruan, Zipeng Zhao, Chen Li, Changbo Wang, and Hong Qin. 2019. A General Novel Parallel Framework for SPH-centric Algorithms. *Proc. ACM Comput. Graph. Interact. Tech.* 2, 1, Article 7 (May 2019), 16 pages. https://doi.org/10.1145/3321360

*Corresponding author.

Authors' addresses: Kemeng Huang, East China Normal University, Shanghai, China, 51174500023@stu.ecnu.edu.cn; Jiming Ruan, East China Normal University, Shanghai, China, ruanjmruanjm@hotmail.com; Zipeng Zhao, East China Normal University, Shanghai, China, zzpzhao@outlook.com; Chen Li, East China Normal University, Shanghai, China, lichen2014gyx@163.com; Changbo Wang, East China Normal University, Shanghai, China, cbwang@sei.ecnu.edu.cn; Hong Qin, Stony Brook University, New York, United States, qin@cs.stonybrook.edu.

2577-6193/2019/5-ART7 $15.00
https://doi.org/10.1145/3321360

1 INTRODUCTION AND MOTIVATION

SPH is a popular mesh-free Lagrangian-based numerical method, which has been widely applied in fluid simulation, astrophysics, and bio-simulation in recent decades. As a particle-based approach, SPH must call for a large number of particles for high-fidelity numerical simulation with fine-scale details and/or large-scale scenes, resulting in a significant cost increase of computational resources. Therefore, it is of great significance to continue to improve the computational performance of SPH and its variants. One key consideration for users is to seek a right tradeoff between simulation effects and system efficiency in practice.

So far, much research has been focusing on the aspect of performance improvement. One major theme in current research aims to improve the performance from the perspective of system architecture. Because of their adaptivity to parallelism in SPH and its variants, some research in recent years directly benefits from the emergence of new hardware platforms and software development toolkits. Yet, much research, especially those making use of GPUs, has not yet taken full advantage of currently-available powerful hardware systems. For example, one popular traditional GPGPU methods (TRA) [Crespo et al. 2015] [Hérault et al. 2010] can significantly improve the performance compared with corresponding CPU implementation. However, to certain extent, they still regard GPU as a *single instruction, multiple threads* (SIMT) processor and the characteristics of thread and memory hierarchy in GPU are not fully utilized. Such under-utilization limits the performance of their implementations by memory bandwidth. To ameliorate, Goswami et al. first proposed a shared memory based method (PSMS) [Goswami et al. 2010]. But their method does not achieve a better performance than TRA, which results from less optimal use of several GPGPU characteristics.

In this paper, our goal is to break the performance bottleneck in the existing SPH implementations. Our novel framework is completely designed for GPGPU architecture. In this framework, we first devise a newly designed parallel task assignment algorithm, which considers the distribution of the particles and the GPU's schedule mechanism, with a goal of achieving a higher cache hit rate. In addition, we design a novel memory access pattern based on the characteristics of memory hierarchy on GPU, since the bandwidth of shared memory is much higher than global memory. Also, we leverage several technologies and methodologies about taking multiple factors into consideration, including the efficiency of thread cooperation, the balance of memory access and the influence from device occupancy. Furthermore, we design a thread warp based radical thread organization. Our key contributions could be highlighted as follows:

- We develop a new parallel task assignment algorithm following the manner of CTA scheduler, so as to achieve higher cache hit rate and reduce the time consumption for threads synchronization, which takes full advantage of shared memory.
- We develop an efficient neighbor traversal algorithm based on the characteristics of memory hierarchy on GPU.
- We develop a hybrid schedule algorithm by a new hash coding with the consideration of particles' arbitrary distribution in space, so as to integrate the traditional neighbor traversal method into our framework.

2 RELATED WORKS

As a flexible and powerful method (with mass-conservation property), SPH has been widely used in fluid simulations [Müller et al. 2003] [Becker and Teschner 2007] [Ihmsen et al. 2014b], as well as deformable solid simulations [Libersky and Petschek 1991] [Müller et al. 2004]. GPU implementations for SPH afford a real-time simulation. However, the current challenge is still hinging upon the high

demand for more scene details while still enhancing computational performance. Current research thrusts are concentrating on several perspectives with an ultimate goal of better performance.

From the perspective of system architectures, some research focuses on this aspect. In [Ihmsen et al. 2011], Ihmsen et al. proposed a sophisticated SPH implementation on CPUs. However, SPH implementations on CPUs have become less competitive due to the rapid development of GPGPU in recent years. Hérault et al. proposed one of the earliest SPH implementations [Hérault et al. 2010] based on the CUDA model. With the help of *Particles*, the example in the CUDA toolkit [Green 2008], Hérault's full GPU implementation can exhibit one to two orders of magnitude faster than the equivalent CPU implementation. The GPU part of DualSPHysics [Crespo et al. 2015] inherits the idea in [Hérault et al. 2010] with several optimization suggestions stated in [Domínguez et al. 2013a]. Goswami et al. introduced an innovative GPU-based implementation [Goswami et al. 2010] for SPH. They tried to make use of the thread and memory hierarchy in GPGPU to improve performance.

Due to the limitation of the computing power of a single device, many researchers took the distributed system into consideration. Homogeneous multi-GPU platform is the simplest way to utilize multiple devices like [Valdez-Balderas et al. 2013]. Rustico et al. also proposed a multi-GPU edition for SPH implementation [Rustico et al. 2014] based on [Hérault et al. 2010] with a posteriori load balancing system for handling different workload among devices. Domínguez et al. extended [Valdez-Balderas et al. 2013] to heterogeneous multi-GPU clusters [Domínguez et al. 2013b].

From the algorithmic perspective, some research focuses on improving the SPH algorithms for better performance. Ihmsen et al. detailed the uniform grid method in [Ihmsen et al. 2011]. This method has been widely used in many SPH implementations for neighbor search. Hoetzlein proposed an improvement [Rama C. Hoetzlein 2014] in generating the neighbor list of the uniform grid method. Mokos et al. developed a new GPGPU-based framework [Mokos et al. 2015] to accelerate multi-phase SPH simulation (involving fluid and air). Yang et al. designed a unified particle system framework [Yang et al. 2017] to accelerate multi-material visual simulations.

The arbitrary distribution of the particles results in the number of the particles in each subspace becoming less controllable, yet these subspaces are used to help search the neighbor particles and called cells in this paper. In the neighborhood grid method [Joselli et al. 2015] and GROMACS [Hess et al. 2008], which is a method in the field of molecular dynamics (MD), the number of particles in each cell could be the same according to some adjustment in the cell boundary. Thus the interaction between particles can be greatly simplified. Páll and Hess also proposed a newly designed algorithm [Páll and Hess 2013] which is specific to the above-mentioned optimization on the SIMD architectures for acceleration.

From the visual effect perspective, some other improvements of SPH sacrifice correctness for the purpose of better efficiency, so this tradeoff is suitable for applications with less strict requirement on correctness, such applications include video games and digital movies. Typical techniques may include, applying different densities of particles for both performance and surface details [Horvath and Solenthaler 2013] [Orthmann and Kolb 2012] [Yan et al. 2009], data-driven method [Ladicky et al. 2015], etc.

3 METHOD OVERVIEW

In this section, we present an overview for our implementation of SPH, a Lagrangian-based method, which discretizes continuous fields by means of particles. Continuous properties of particles are approximated by weighted interpolation over smoothed functions $W(\mathbf{r}, h)$ [Monaghan 1992] [Ihmsen et al. 2014b]:

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

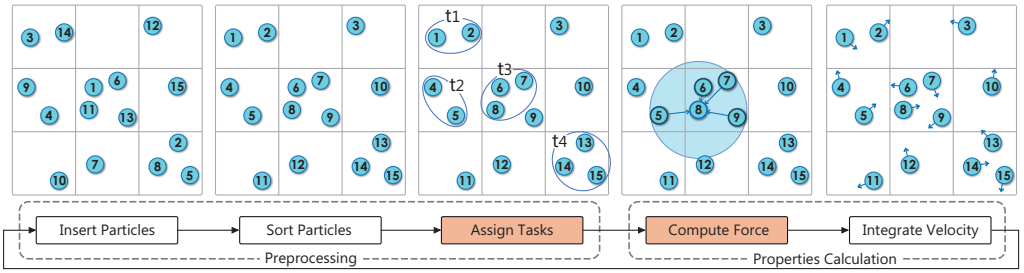

Fig. 1. The framework of the SPH implementation. The red blocks are the major improvements in this paper.

where $A(\mathbf{r}_i)$ is some continuous variable of the particle at position $\mathbf{r}_i$, such as density or force, m and ρ refer to mass and density. Smoothing radius h in $W(\mathbf{r}, h)$ defines the influence space within which the contribution from the rest particles should be collected. So the neighbor traversal problem of SPH can be mapped to the *fixed-radius near neighbor* (FNN) problem. Considering adaptivity to parallelism and flexibility in numerical methods, we use a uniform grid method to solve such problem (see Fig. 1). In this method, the simulation domain is partitioned into a set of non-intersecting indexed cells $\mathbb{C}$, whose size equals to smoothing radius h. Each particle is inserted into one cell. Thus, during the neighbor traversal step, neighbor particles of cell $\mathbf{C_J}$ are sought only by cell set $\mathbb{C}_\mathbf{J}$, which is the collection of the neighbor cells of $\mathbf{C_J}$, $\mathbf{J}$ representing the spatial position of $\mathbf{C_J}$ in Euclidean space. The procedure of computation has two steps: neighbor information generation, and neighbor traversal. The time complexity of the former is $O(mn)$ and the cost of neighbor traversal is $O(3^k mnN)$ where m is the hash value size, n is the number of particles in space, k is the dimension of space, and N is the upper bound of the particles in each cell [Bentley et al. 1977].

In our implementation, the uniform grid is defined as its range in Euclidean space and cell size. Since there is no data dependency among particles, we can calculate the cell indices (hash value) for each particle using the definition of grid and particle position in parallel (cell indices are calculated in the x-axis-first manner). Then, the GPU counting sort algorithm [Rama C. Hoetzlein 2014] is used to sort particles according to cell indices. After sorting, the data of the particles existing in the same cell and adjacent cells (the cells with the same y- and z-axis position, and adjacent x-axis position) are continuous (without gap) in memory. Meanwhile, we can get the count array $\mathbf{B}$ and offset array $\mathbf{O}$ of particles for each cell, and $\mathbf{B}$ will be used in particles task assignment and classification. We will detail these parts in Section 4.1 and Section 4.3. It may be noted that, $\mathbf{B}$ and $\mathbf{O}$ are also regarded as the neighbor list for each cell here. In our implementation, there is no neighbor list for each particle. So in the properties calculation stage (see Fig. 1), the neighbor sets are the supersets of the sets of particles whose distances are small than h. Based on our schedule assignment strategy, we design an efficient shared memory based method to compute continuous variable of particles. We will detail this part in Section 4.2.

4 NOVEL GENERAL PARALLEL METHOD

In this section, we will first detail our task assignment algorithm. Then the efficient neighbor traversal method with a well-designed memory and thread allocation strategy is introduced. Finally we will detail our new hash coding method and our hybrid strategy.

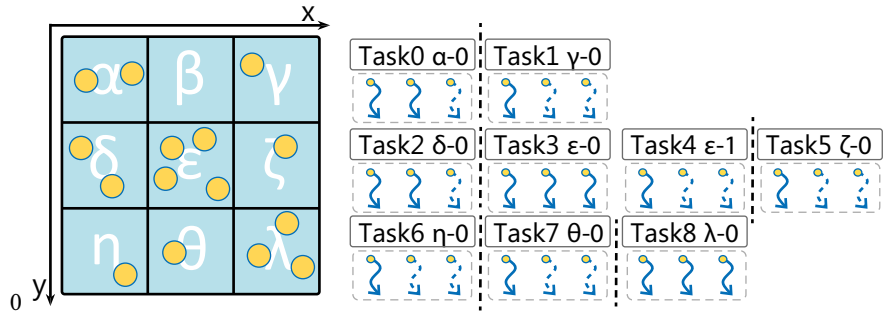

Fig. 2. The schematic diagram about how the tasks from each cell are assigned to the CTAs. The dotted lines represent the threads which do not compute properties for each particle.

4.1 Task Assignment

In GPGPU architectures, thread scheduling has a significant impact on overall performance. In order to achieve high parallelism and take full advantages of available resources, a CTA-based thread scheduling strategy is used in our framework. Since the number of threads (D) in each CTA is constant while the number of particles in each cell is variable, task assignment is necessary when assigning particles into CTAs. Fig. 2 shows the result of task assignment in a simulation domain partitioned by a uniform grid. The number of tasks $\mathbf{B}'_j$ in cell $\mathbf{C}_j$ is equal to $\lceil \mathbf{B}_j/D \rceil$ (in this figure, we assume $D = 3$, j is the index of cell). We assign a CTA to each task, which means there are no multiple tasks sharing the same CTA. After calculating the number of tasks $\mathbf{B}'$ for cells, we should arrange these tasks into task array $\mathbf{Q}$ according to the definition of $\mathbf{Q}$, which has following form:

$$\mathbf{Q} = \{t_0, t_1, t_2, \cdots, t_{S-2}, t_{S-1}\}, \tag{2}$$

$$t_i = \{\langle j_i, k_i \rangle | \mathbf{C}_{j_i} \in \mathbb{C}, k_i \in \mathbf{Z}_0^+, 0 \le i \le S-1\}, \tag{3}$$

where k_i is the index of task in cell $\mathbf{C}_{j_i}$. For example (see Fig. 2), the count of all the CTAs $S = 9$ and $t_4 = \langle \varepsilon, 1 \rangle$ in the x-axis-first manner. Thereby, each CTA can locate its task quickly according to task t_i. Fig. 6 shows the relationship between thread block and CTA (each thread block contains 2 CTAs, so the count of all the thread blocks is equal to $\lceil S/2 \rceil$), we locate a CTA for each task according to the following form:

$$i = 2 \times Bid + a, \tag{4}$$

where Bid is the indices of thread block, a is the index of CTA in the thread block.

Fig. 3 gives an overview of our task assignment. First, we use $|\mathbf{C}|$ threads in GPU to compute $\mathbf{B}'$ for cells based on the rules stated above. Second, we use an efficient GPGPU-based prefix sum algorithm on $\mathbf{B}'$ to calculate the task offset $\mathbf{O}'$ of cells. Third, $|\mathbf{C}|$ threads in GPU are used to arrange tasks into $\mathbf{Q}$ based on $\mathbf{B}'$ and $\mathbf{O}'$. Finally, we obtain S by adding the last value in $\mathbf{O}'$ and the last non-zero value in $\mathbf{B}'$. The implementation of the task assignment is summarized in Alg. 1 (array $\mathbf{c}$ are the indices of particles in a cell).

Our task assignment always guarantees the sequence of tasks. In PSMS, task assignment is designed based on atomic operations. This method can't guarantee the sequence of tasks, because the return values of atomic operations are random. So the adjacent tasks may have large distance in task array. On the other hand, the CTA scheduler in CUDA model follows an approximate round-robin manner [Lee et al. 2014], which means the adjacent CTAs are more likely to be executed simultaneously. In addition, if the data accessed by adjacent CTAs are continuous in memory, we can get a higher cache hit rate, especially for L2 cache [Wittenbrink et al. 2011]. Therefore our method can achieve better performance compared with PSMS.

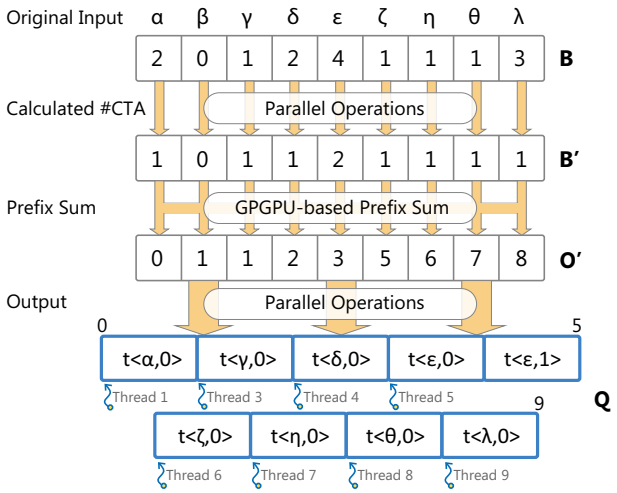

Fig. 3. The overview of the task assignment processes. The original input is the array **B** of particle count in cells. The output is the array **Q** of task.

Algorithm 1 Task Assignment.

//GPU counting sort

1: **B**, **c** ← conduct CUDA ***atomicAdd*** operations on particles' hash values
2: **O** ← conduct GPU-based prefix sum algorithm on **B**
3: sort particles according to **B**, **O**, **c**

//task assignment

4: **B**′ ← $\lceil \mathbf{B}/D \rceil$
5: **O**′ ← conduct GPU-based prefix sum algorithm on **B**′
6: **Q** ← arrange tasks into **Q**

4.2 Neighbor Traversal

The novel neighbor traversal algorithm is the main innovation of our framework. In our algorithm, we try to use several advanced characteristics of GPGPU architectures effectively.

Fig. 4(a) shows the main procedures of TRA and our method. The main procedures of TRA, PSMS, and our method are the same: (1) fetch the data of the target particles from global memory to the on-chip memory; (2) enter a cycle that fetches the data of one or multiple neighbors from global memory to the on-chip memory; (3) conduct calculation for target particles; (4) send the results back to the global memory. The main difference between TRA and our method is the usage of the shared memory. Compared with fetching one neighbor particle to registers for each thread in TRA, our method fetches at most 32 neighbor particles to the shared memory for each CTA (one neighbor particle per thread, $D = 32$ in our framework). As the particles handled by the same CTA belong to the same cell, these particles always share the same neighbor particles. In this way, the fetched 32 neighbor particles can be used for all threads in the CTA, and the cost of shared memory access is much lower than global memory. Therefore our method can greatly reduce requirements of registers and requests of global memory when fetching the neighbor particles. And Fig. 4(b) shows the evident differences with a simple example.

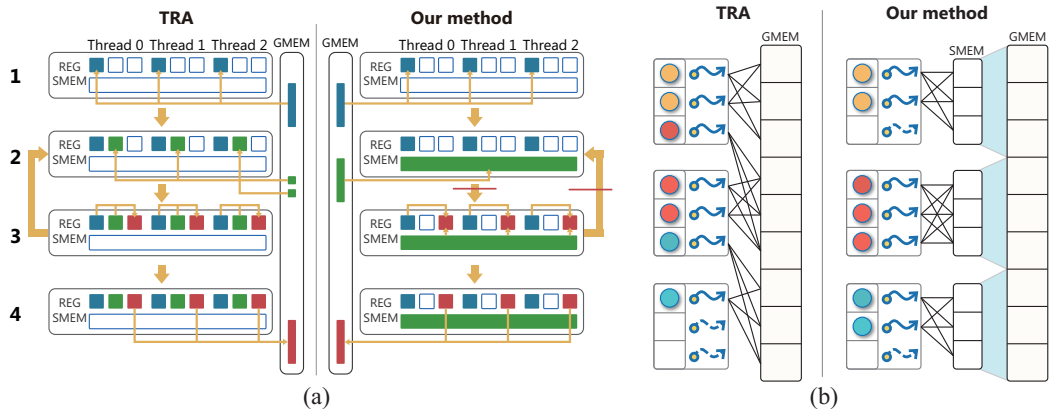

Fig. 4. Figure (a) shows the procedures of neighbor traversal and attribute computation in TRA (left) and our method (right). Blue blocks refer to the data from the particles needed to be computed, the green blocks refer to the data from neighbors, and the red blocks refer to the resulting data. The red lines refer to CTA synchronization. REG is register. SMEM is shared memory. GMEM is global memory. Figure (b) shows the differences of memory operation in TRA (left) and our method (right). The circles shown with different colors, represent the particles in different cells of the grid.

In PSMS, each thread block fetches at most 27 neighbor particles (i.e., 27 threads fetch 27 neighbor particles from different neighbor cells). This method involves a lot of memory transactions while our method can greatly reduce it. During each memory request in our method, the 32 requested neighbor particles always come from the same cell or adjacent cells. Since these particles are always stored in continuous memory, the memory transactions from the same memory request can be combined in most cases. Thus, the cost of global memory access can be greatly reduced.

Alg. 2 shows the details of our neighbor traversal algorithm. Lines 2-4 are procedures of assigning CTA for each task, which involve the calculations of a and q (indices of thread in CTA). After that, we can identify active threads and idle threads (line 5). Each active thread fetches target particle from global memory to registers (lines 6-8). In addition, the indices of neighbor cells are determined in a uniform grid, because we have calculated the indices of target cell. Thereby, the ranges of neighbor particles are determined. Here, we apply the optimization called *simplifying the neighbor search* proposed in [Domínguez et al. 2013a]. This optimization regards three adjacent cells as a cuboid cell, so as to reduce memory transactions and shared memory requests. Therefore the number v of all the neighbor cells is set to 9 rather than 27, which means we compress 27 cubic neighbor cells into 9 cuboid cells. We store the ranges of these 9 cuboid cells as offset array $\mathbf{o}$ and count array $\mathbf{b}$ in shared memory (lines 9-13) according to $\mathbf{B}, \mathbf{O}$. Line 10 is the calculation of the correct index e for $\mathbf{o}$ and $\mathbf{b}$, because a thread block contains 2 CTAs (see Fig. 10). At the end of initialization, array $\mathbf{f}$ and $\mathbf{u}$ are initialized by threads (line 14). $\mathbf{f}_d$ represents the index of the cuboid neighbor cell, which is being accessed by thread d. $\mathbf{u}$ represents the count of neighbor particles, which have been accessed in cuboid cell $\mathbf{f}_d$. Both arrays reside in shared memory.

After the initialization of neighbor information (lines 9-14), the rest procedures are the migration of data (from global memory to shared memory) and the calculation of variables. At first, lines 16-21 is the judgment of whether all the neighbor particles in cuboid cell $\mathbf{f}_d$ have been accessed (line 16). If so, switch to the next cuboid cell or end the loop (lines 17-20). Then, at most D particles are fetched from global memory to shared memory for each CTA (lines 22-24). It is noteworthy that only the needed data are read. Since the variables of the particles are stored as a structure in corresponding

Algorithm 2 Novel Neighbor Traversal.

```
1: for all threads do
   //CTA location and self-data load
2:     //a is the indices of CTA in thread block and q is the indices of thread in CTA
3:     a ← ⌊d/D⌋, q ← d%D
4:     i ← compute the indices of task using Eq. 4.
5:     j, k ← get the indices of cell and indices of CTA in the cell from Q_i
6:     tr ← B_j + k × D + q //tr is the indices of particle in particle array
7:     if tr < B_j + O_j then
8:         read particle tr from GMEM to REG
9:     end if
   //neighbor information initialization
10:    if q < v then //v is count of cuboid neighbor cell
11:        //get corresponding neighbor range in GMEM
12:        e ← a × v + q
13:        o_e ← particle offset of cuboid neighbor cells
14:        b_e ← particle count of cuboid neighbor cells
15:    end if
16:    syncthreads
17:    f_d ← a × v, u_d ← 0 //initial the accessing cuboid cells information
   //main loop
18:    while true do
19:        //determine whether all neighbors in cuboid cell f_d are accessed
20:        if u_d == b_{f_d} then
21:            f_d ← f_d + 1, u_d ← 0 //update accessing cuboid cell information
22:            if v × (a + 1) ≤ f_d then //determine whether neighbor accessing is end
23:                break
24:            end if
25:        end if
26:        if q < min(D, b_{f_d} − u_d) then
27:            pi ← o_{f_d} + u_d + q //calculate the indices of neighbor in GMEM
28:            read particle pi from GMEM to SMEM
29:        end if
30:        u_d ← u_d + min(D, b_{f_d} − u_d)
31:        syncthreads
32:        if tr < B_j + O_j then
33:            calculate variables of target particle
34:        end if
35:    end while
36: end for
```

arrays, this method can effectively reduce the demand for memory requests. At last, each active thread makes use of the variables of target particle stored in the registers and neighbor particles stored in shared memory to conduct the calculation of variables (lines 28-30).

In our framework, D is set to 32, which means there are 32 threads in each CTA and 64 threads in each thread block (the number of CTA slots and warp slots supported by each SM in the recent NVIDIA GPUs are 32 and 64 [Li et al. 2017], so our device occupancy can be 100%). Because of the

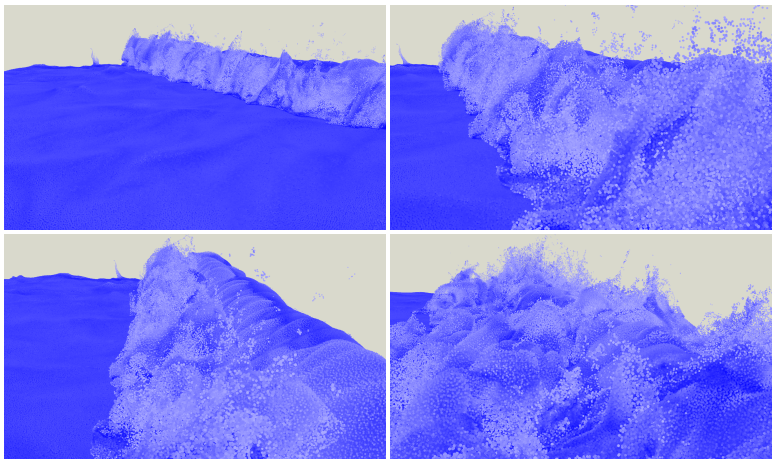

Fig. 5. From top to bottom, left to right: ocean wave with 18,440,253 particles.

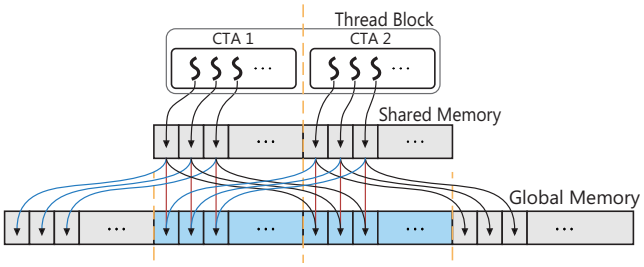

Fig. 6. The overview of memory operations in a thread block. The blue blocks represent the common memory part of neighbor particles. Arrows represent the direction of memory requests. The lines, between shared memory and global memory, shown with different colors, represent the order of memory request. Blue first, red second, and black third.

native synchronous execution of threads in a warp, our setting greatly reduces the time of thread synchronization operations (we will validate this design in Section 5.1). Moreover, two-CTAs-based task assignment can further improve the cache hit rate, as two adjacent CTAs often share a common part of neighbor particles (see Fig. 6).

4.3 Hybrid Strategy

In most cases, our task assignment and neighbor traversal methods have good performance. However, our methods may lead to some unnecessary waste of resources, as the size of CTA is constant. For example, in Fig. 2, there is only one particle in cell γ. So one thread is enough to deal with the particle while the other threads in CTA are idle. Moreover, in this cell, fetched neighbor particles are not shared with multiple active threads and the number of neighbor particles is relatively small. Therefore, in this case, our method will result in poor performance. The similar problem may exist in the 5th CTA, as we assign a CTA to the 4th particle of $\mathbf{C}_\varepsilon$ (such particles are named sparse particles while other particles are named intensive particles in this paper). In order to avoid these problems, the distribution of particles should be considered. Hence we propose a new hash coding method, whose hash function maps position $\mathbf{P} = (x, y, z)$ to a hash value of size $2m$. The function

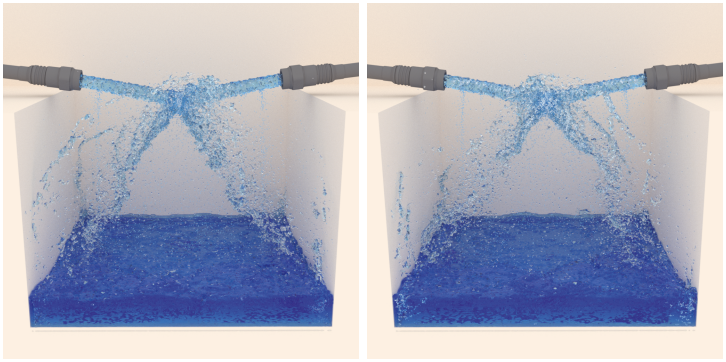

Fig. 7. Two water jet streams collide with each other.

has the following form:

$$H(\mathbf{P}) = \begin{cases} \lfloor \frac{x}{h} \rfloor + \lfloor \frac{y}{h} \rfloor \cdot X + \lfloor \frac{z}{h} \rfloor \cdot X \cdot Y + m, & T = 1, \\ \lfloor \frac{x}{h} \rfloor + \lfloor \frac{y}{h} \rfloor \cdot X + \lfloor \frac{z}{h} \rfloor \cdot X \cdot Y, & T = 0, \end{cases} \tag{5}$$

where X, Y are the numbers of cells of the uniform grid in x- and y-axis. The value of T is 0 or 1, which depends on $\mathbf{c}$ and $\mathbf{B}$. We can get $\mathbf{c}$ by ***atomicAdd*** operations in CUDA model, and $\mathbf{B}'$ is calculated with a new function:

$$\mathbf{B}'_j = \begin{cases} 0, & \overline{\mathbf{B}_j} < P_{of}, \\ \lfloor (\mathbf{B}_j + N_{of})/D \rfloor, & \overline{\mathbf{B}_j} \geq P_{of}, \end{cases} \tag{6}$$

where N_{of} is a threshold, which determines the maximum number of idle threads in CTA, $\overline{\mathbf{B}_j}$ is the average particles of $\mathbf{C}_j$ (we only consider target cell and 6 nearest neighbor cells, and divide summation of the particle counts by 7 to get a approximate average count). And P_{of} is the threshold of the minimum average particles in target cell, which determines whether assigns CTAs to target cell or not. Once $\mathbf{B}'_j$ is calculated, T has the following form:

$$T = \begin{cases} 1, & \mathbf{B}'_j \cdot D \geq \mathbf{c}, \\ 0, & \mathbf{B}'_j \cdot D < \mathbf{c}. \end{cases} \tag{7}$$

Based on Eqs. 5, 6, and 7, the new hash value of $\mathbf{P}$ can be determined. For example, $\mathbf{B}_\varepsilon = 4$, in $\mathbf{C}_\varepsilon$ (see Fig. 2), $\mathbf{B}_\varepsilon{}' = 1$ (we set $N_{of} = 1$ and $P_{of} = 2$) according to Eq. 6. The hash value of 4th particle is 4 while others are 13 according to Eqs. 5, and 7 (m is equal to 9 in Fig. 2). Similarly, the hash value of particle in $\mathbf{C}_\theta$ is equal to 7. And we will find that the hash value of sparse particles is smaller than 9 while others are no less than 9. Hence, after sorting particles based on the new hash value again, particles are divided into sparse particles group and intensive particles group.

As most sparse particles are not in the same cells, shared memory may not be helpful, even result in poor performance. Therefore we integrate the traditional neighbor traversal method into our framework to deal with sparse particles. When we launch thread blocks, we arrange the front part of thread blocks to deal with sparse particles and regard the number of these thread blocks as a threshold, which helps thread blocks distinguish the different particle groups (see Fig. 8, the threshold is equal to 1). This hybrid strategy can reduce the thread waste and unreasonable memory operations, especially when the particles are sparse. The implementation of our reasonable hybrid strategy is summarized in Alg. 3.

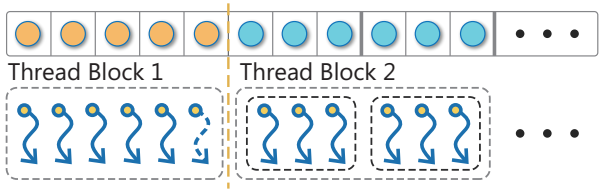

Fig. 8. The yellow circles represent sparse particles and the blue circles represent intensive particles. We assign a thread to each sparse particle and assign a CTA to each task.

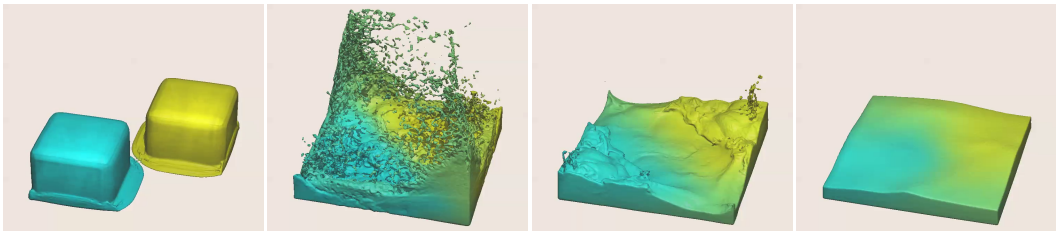

Fig. 9. The interaction between two different miscible fluids.

Algorithm 3 Hybrid Framework.

repeat
*//**preprocessing***
  $\mathbf{B}, \mathbf{O}, \mathbf{c} \leftarrow$ conduct GPU counting sort in Alg. 1
  $\mathbf{B}' \leftarrow$ calculate $\mathbf{B}'$ and new hash value using Eq. 5, 6, 7
  conduct GPU counting sort on new hash values.
  $B_v \leftarrow$ find the boundary value of particles groups
  $\mathbf{O}' \leftarrow$ conduct GPU prefix sum method on $\mathbf{B}'$
  $\mathbf{Q} \leftarrow$ organize task array $\mathbf{Q}$
*//**variables calculation***
  if thread block belong to sparse particles **then**
    for each sparse particle i **do**
      read particle i from *global memory* to *registers*
      calculate variables of particle i
    end for
  else
    use Alg. 2 to deal with *intensive particles*
  end if
until end of simulation

5 RESULTS AND EVALUATIONS

In this section, we evaluate our novel parallel framework using several scenes, which are implemented with different SPH algorithms. The experiments are performed on a system with an Intel(R) Core(TM) i5 4590 and NVIDIA Geforce GTX 970.

We compare our framework with two different frameworks (PSMS and TRA). PSMS is proposed in [Goswami et al. 2010]. TRA is first proposed in [Hérault et al. 2010], and the similar method is used in DualSPHysics [Crespo et al. 2015]. In TRA, there is a one-to-one relationship between a

Table 1. Test case settings and simulation results. The results are the average of the first 1000 time steps. **PPNC** is the average particles of nonempty cells. **Pre** is preprocessing overhead. **Den** is density computation overhead. **Forc** is force computation overhead. **Tot** is total simulation overhead excluded the velocity integration.

#	#particles	PPNC	TRA (ms)				PSMS (ms)				OUR METHOD(ms)			
			Pre	Den	Forc	Tot	Pre	Den	Forc	Tot	Pre	Den	Forc	Tot
case 1	592,704	4.7	**2.2**	3.8	5.7	11.7	3.3	10.8	20.1	34.2	4.5	**1.3**	**2.6**	**8.4**
case 2	1,000,000	8.0	**3.0**	6.1	10.1	19.2	3.8	14.1	28.1	46.0	5.6	**3.5**	**6.7**	**15.8**
case 3	1,815,848	14.5	**3.5**	15.0	33.4	51.9	5.7	34.0	68.5	108.2	6.8	**10.6**	**23.4**	**40.8**
case 4	3,652,264	29.2	**4.7**	49.0	112	165.7	8.0	67.4	141.9	217.3	10.1	**32.3**	**63.2**	**105.6**
case 5	7,189,057	57.5	**12.5**	169.8	400	582.3	14.1	205.8	444.8	664.7	18.2	**101.5**	**208.9**	**328.4**
case 6	13,481,272	107.9	**33.5**	565.5	1299.5	1898.5	33.7	753.2	1562.9	2349.8	42.0	**369.5**	**736.5**	**1148.0**

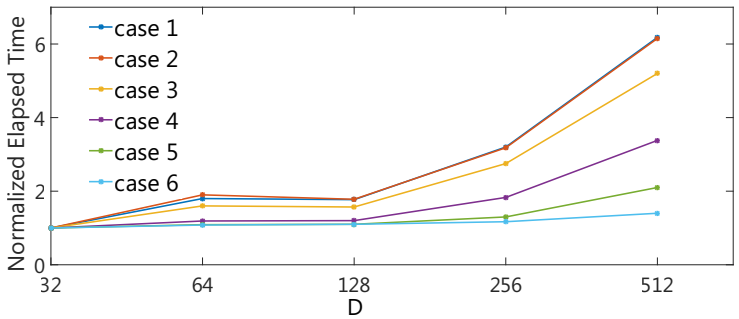

Fig. 10. Average elapsed time using different D settings. Time values are normalized to the cases with $D = 32$.

thread and a particle. Due to its simple manner and good performance, TRA becomes a popular GPGPU method for SPH.

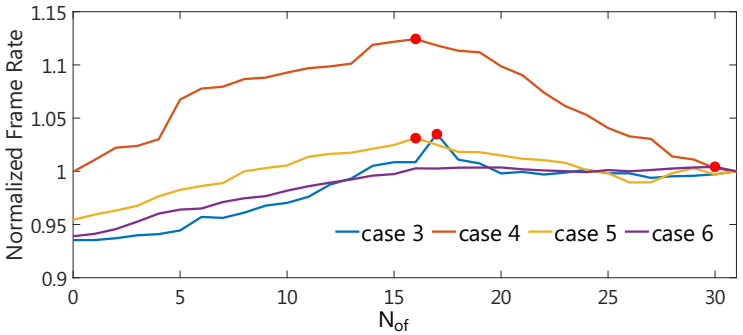

Fig. 11. Average frame rate using different N_{of} settings. Frame rates are normalized to the cases with $N_{of} = 31$.

5.1 Parameter Effects

First of all, we explore the nature of our exposed parameters to find the correlations between their configuration and the resulting performance. For brevity, the benchmark used here is the collapse of cuboid fluid with the basic SPH. The key information of the test cases and detailed elapsed time are shown in Table 1. These test cases are different not only in particle count but also in particle

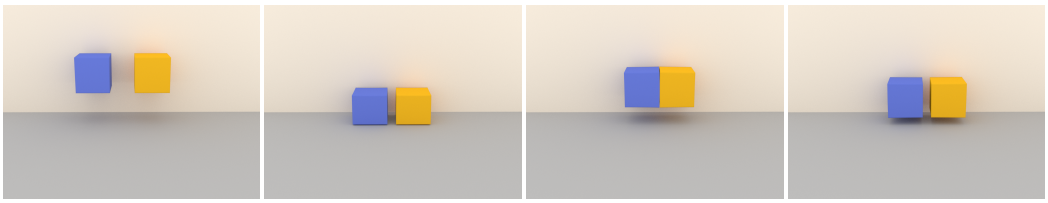

Fig. 12. Two dropping elastic blocks collide with each other.

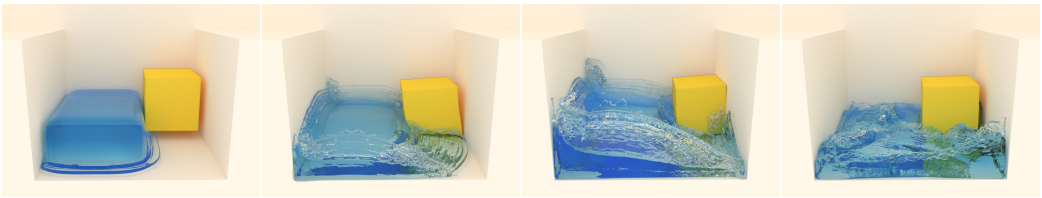

Fig. 13. The interaction between elastic block and fluid.

intensity. Higher particle intensity will lead to more memory requests, which significantly increases the cost of computations. This table shows the evident improvement of our framework, which can achieve 4× speedup compared with PSMS. Moreover, the table shows that the preprocessing overhead of TRA is always the smallest owing to the simplest preprocessing operation in TRA.

Fig. 10 shows the performance of different CTA size in our framework. CTA synchronizations are added to the cases with $D \geq 32$ for thread safety. The test results show that the cases with $D = 32$ can always achieve the best performance. However, the improvement is gradually weakened with the increase of particle intensity, because higher particle intensity leads to less CTA synchronizations and idle threads relatively.

Through our tests, our framework can always exhibit good performance when P_{of} is equal to 11.86. The appropriate value of N_{of} depends on the distribution of particles. Fig. 11 shows the test results of cases 3-6 (we choose test cases according to Eq. 6, the average particles in each non-empty cell of cases 1-2 is smaller than P_{of}, so the influence of N_{of} can be omitted). We find that setting N_{of} to 16 can exhibit good performance (cases 3-5). However, in case 6, whose particles are intensive, the optimal N_{of} approaches 31, because the relative number of idle threads becomes smaller, but meanwhile our CTA based task assignment can still achieve good performance in a scene with intensive particles.

5.2 Performance

As the performance of TRA is better than PSMS (see Table 1), we only compare our framework with TRA. First, we apply ocean wave with more than 18 million particles (see Fig. 5), to explore the improvement of our framework for a large-scale scene with fluids. Verified by a series of tests/experiments, our framework achieves up to 1.65× speedup compared with TRA.

To further explore the performance of our framework, we use complex SPH algorithms as the benchmark. PCISPH has great incompressibility due to its correction of particle position by much computation process. Therefore, we apply water jet stream collision based on PCISPH [Solenthaler and Pajarola 2009] (see Fig. 7), as the benchmark. Our framework can exhibit up to 1.58× speedup (see the red line in Fig. 14).

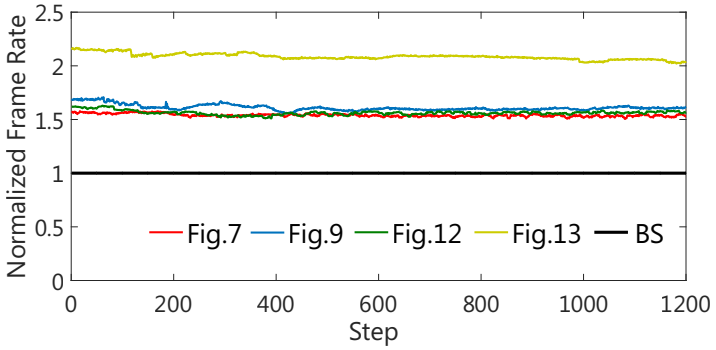

Fig. 14. Benchmark performance results. Fig.X is the performance of the corresponding figure's scene implemented with our framework. BS represents the performance of all benchmarks implemented with TRA in our paper. The frame rate is normalized to the performance of TRA.

Furthermore, we apply multiple fluid SPH (see Fig. 9) [Ren et al. 2014] to explore the performance of our framework with much more complex SPH algorithms. In this example, we achieve a real-time simulation and rendering by our framework with a sparse hierarchy of grids represented in NVIDIA GVDB Voxels [Wu et al. 2018]. Our framework can exhibit up to 1.70× speedup (see the blue line in Fig. 14).

Besides, our framework is applied to solid simulation (e.g. elastic solid) and interaction between elastic solid and fluid [Yan et al. 2016]. Fig. 12 shows the interaction between two dropping elastic blocks. Our framework can get up to 1.63× speedup (see the green line in Fig. 14). Fig. 13 shows the interaction between elastic solid and fluid. Our framework can exhibit up to 2.18× speedup (see the yellow line in Fig. 14).

6 CONCLUSION AND FUTURE WORKS

This paper has devised a novel well-designed SPH framework. This framework makes full use of several characteristics of GPGPU and attempts to combine such features in an optimal way so that great performance improvement is achieved without the need of any modification in the existing numerical method.

The framework designed in this paper can also be readily transplanted to other particle-based methods [Bender and Koschier 2015] [Ihmsen et al. 2014a] [Macklin and Müller 2013] without much difficulty. Essentially, the approaches we have proposed in this paper are based on an improved GPGPU algorithm for solving the FNN problem on uniform grids. Thus, we believe that most of the applications referencing the FNN problem such as the access to the leaf cells phase in fast multiple method [Chandramowlishwaran et al. 2010] [Yokota et al. 2013], point cloud problem, marching cube and astrophysics simulation can directly benefit from our work. On the other hand, we need a good dynamic parameter setting strategy to cater to the distribution of particles in a uniform grid. In addition, thanks to the earlier work in [Orthmann and Kolb 2012], our framework can further make it possible to parallelize the cumulative summation process of SPH, so as to achieve an even greater acceleration.

ACKNOWLEDGMENTS

The authors would like to especially thank all reviewers for their sincere and thoughtful suggestions. This work was supported by National Natural Science Foundation of China under Grants (No. 61672237, 61532002).

REFERENCES

Markus Becker and Matthias Teschner. 2007. Weakly compressible SPH for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2007, San Diego, California, USA, August 2-4, 2007*. 209–217.

Jan Bender and Dan Koschier. 2015. Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation, SCA 2015, Los Angeles, CA, USA, August 7-9, 2015*. ACM, 147–155.

Jon Louis Bentley, Donald F. Stanat, and E. Hollins Williams Jr. 1977. The Complexity of Finding Fixed-Radius Near Neighbors. *Inform. Process. Lett.* 6, 6 (1977), 209–212.

Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, Ilya Lashuk, George Biros, and Richard W. Vuduc. 2010. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 1–12.

Alejandro J. C. Crespo, José M. Domínguez, Benedict D. Rogers, Moncho Gómez-Gesteira, Stephen M. Longshaw, Ricardo B. Canelas, Renato Vacondio, A. Barreiro, and O. García-Feal. 2015. DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH). *Computer Physics Communications* 187 (2015), 204–216.

José M. Domínguez, Alejandro J. C. Crespo, and Moncho Gómez-Gesteira. 2013a. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications* 184, 3 (2013), 617–627.

José M. Domínguez, Alejandro J. C. Crespo, Daniel Valdez-Balderas, Benedict D. Rogers, and Moncho Gómez-Gesteira. 2013b. New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications* 184, 8 (2013), 1848–1860.

Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. 2010. 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, 55–64.

Simon Green. 2008. Cuda particles. *NVIDIA Whitepaper* 2, 3.2 (2008), 1.

Alexis Hérault, Giuseppe Bilotta, and Robert A Dalrymple. 2010. Sph on gpu with cuda. *Journal of Hydraulic Research* 48, S1 (2010), 74–79.

Berk Hess, Carsten Kutzner, David Van Der Spoel, and Erik Lindahl. 2008. GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation* 4, 3 (2008), 435–447.

Christopher Jon Horvath and Barbara Solenthaler. 2013. Mass preserving multi-scale SPH. *Pixar Technical Memo* 13-04 (2013).

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. 2011. A Parallel SPH Implementation on Multi-Core CPUs. *Computer Graphics Forum* 30, 1 (2011), 99–112.

Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christopher Horvath, and Matthias Teschner. 2014a. Implicit Incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (2014), 426–435.

Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. 2014b. SPH Fluids in Computer Graphics. In *Eurographics 2014 - State of the Art Reports, Strasbourg, France, April 7-11, 2014*. ACM, 21–42.

Mark Joselli, José Ricardo da S. Junior, Esteban Walter Gonzalez Clua, Anselmo Antunes Montenegro, Marcos Lage, and Paulo A. Pagliosa. 2015. Neighborhood grid: A novel data structure for fluids animation with GPU computing. *J. Parallel and Distrib. Comput.* 75 (2015), 20–28.

Lubor Ladicky, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus H. Gross. 2015. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics* 34, 6 (2015), 199:1–199:9.

Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE, 260–271.

Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. 297–311.

Larry D Libersky and Albert G Petschek. 1991. Smooth particle hydrodynamics with strength of materials. In *Advances in the Free-Lagrange Method Including Contributions on Adaptive Gridding and the Smooth Particle Hydrodynamics Method*. Springer, 248–257.

Miles Macklin and Matthias Müller. 2013. Position based fluids. *ACM Transactions on Graphics* 32, 4 (2013), 104.

Athanasios Mokos, Benedict D. Rogers, Peter Stansby, and José M. Domínguez. 2015. Multi-phase SPH modelling of violent hydrodynamics on GPUs. *Computer Physics Communications* 196 (2015), 304–316.

Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics* 30, 1 (1992), 543–574.

Matthias Müller, David Charypar, and Markus H. Gross. 2003. 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*. 154–159.

Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus H. Gross, and Marc Alexa. 2004. 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*. 141–151.

Jens Orthmann and Andreas Kolb. 2012. Temporal Blending for Adaptive SPH. *Computer Graphics Forum* 31, 8, 2436–2449.

Szilárd Páll and Berk Hess. 2013. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications* 184, 12 (2013), 2641–2650.

Graphics Devtech Rama C. Hoetzlein. 2014. Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. In *GPU Technology Conference*. NVIDIA.

Bo Ren, Chen-Feng Li, Xiao Yan, Ming C. Lin, Javier Bonet, and Shi-Min Hu. 2014. Multiple-Fluid SPH Simulation Using a Mixture Model. *ACM Transactions on Graphics* 33, 5 (2014), 171:1–171:11.

Eugenio Rustico, Giuseppe Bilotta, Alexis Hérault, Ciro Del Negro, and Giovanni Gallo. 2014. Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2014), 43–52.

Barbara Solenthaler and Renato Pajarola. 2009. Predictive-corrective incompressible SPH. *ACM Transactions on Graphics* 28, 3 (2009), 40:1–40:6.

Daniel Valdez-Balderas, José M. Domínguez, Benedict D. Rogers, and Alejandro J. C. Crespo. 2013. Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters. *J. Parallel and Distrib. Comput.* 73, 11 (2013), 1483–1493.

Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU Architecture. *IEEE Micro* 31, 2 (2011), 50–59.

Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast Fluid Simulations with Sparse Volumes on the GPU. *Computer Graphics Forum* 37, 2 (2018), 157–167.

He Yan, Zhangye Wang, Jian He, Xi Chen, Changbo Wang, and Qunsheng Peng. 2009. Real-time fluid simulation with adaptive SPH. *Journal of Visualization and Computer Animation* 20, 2-3 (2009), 417–426.

Xiao Yan, Yun-Tao Jiang, Chen-Feng Li, Ralph R. Martin, and Shi-Min Hu. 2016. Multiphase SPH simulation for interactive fluids and solids. *ACM Transactions on Graphics* 35, 4 (2016), 79:1–79:11.

Tao Yang, Jian Chang, Ming C. Lin, Ralph R. Martin, Jian J. Zhang, and Shi-Min Hu. 2017. A unified particle system framework for multi-phase, multi-material visual simulations. *ACM Transactions on Graphics* 36, 6 (2017), 224:1–224:13.

Rio Yokota, Lorena A. Barba, Tetsu Narumi, and Kenji Yasuoka. 2013. Petascale turbulence simulation using a highly parallel fast multipole method on GPUs. *Computer Physics Communications* 184, 3 (2013), 445–455.