GPU Optimization of Material Point Methods

MING GAO*†, University of Pennsylvania
XINLEI WANG*‡, University of Pennsylvania
KUI WU*, University of Utah
ANDRE PRADHANA, University of Pennsylvania and DreamWorks Animation
EFTYCHIOS SIFAKIS, University of Wisconsin - Madison
CEM YUKSEL, University of Utah
CHENFANFU JIANG, University of Pennsylvania

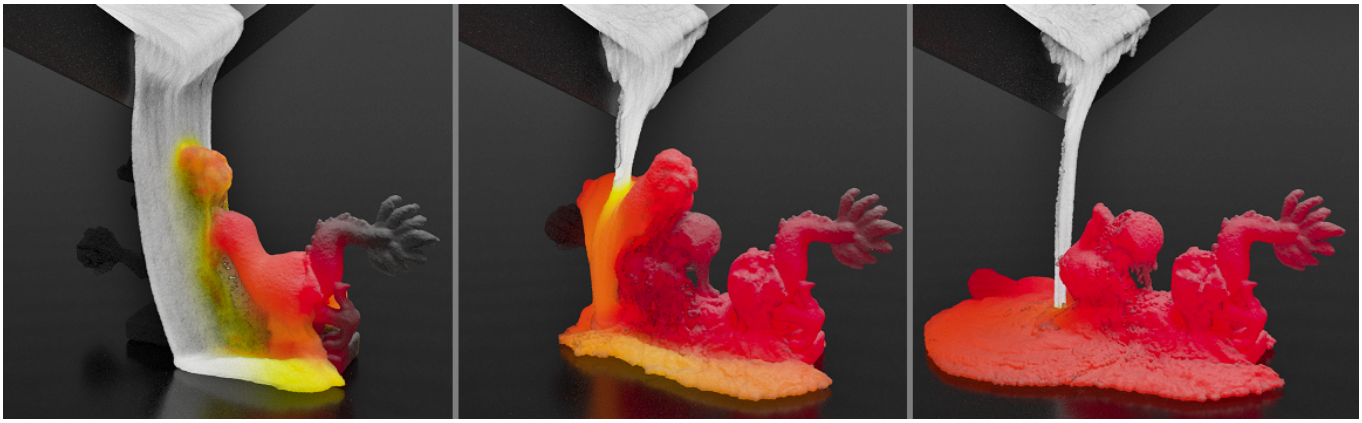

Fig. 1. **How to melt your dragon.** Melting an elastoplastic dragon with 4.2 million particles on a 256^3 grid using our GPU-optimized implicit MPM dynamics and heat solvers on a Nvidia *Quadro* P6000 GPU at an average 10.5 seconds per $48Hz$ frame.

The Material Point Method (MPM) has been shown to facilitate effective simulations of physically complex and topologically challenging materials, with a wealth of emerging applications in computational engineering and visual computing. Borne out of the extreme importance of regularity, MPM is given attractive parallelization opportunities on high-performance modern multiprocessors. Parallelization of MPM that fully leverages computing resources presents challenges that require exploring an extensive design-space for favorable data structures and algorithms. Unlike the conceptually simple CPU parallelization, where the coarse partition of tasks can be easily applied, it takes greater effort to reach the GPU hardware saturation due to its many-core SIMT architecture. In this paper we introduce methods for addressing the computational challenges of MPM and extending the capabilities of general simulation systems based on MPM, particularly concentrating on GPU optimization. In addition to our open-source high-performance framework, we also conduct performance analyses and benchmark experiments to compare against alternative design choices which may superficially appear to be reasonable, but can suffer from suboptimal performance in practice. Our explicit and fully implicit GPU MPM solvers are further equipped with a Moving Least Squares MPM heat solver and a novel sand constitutive model to enable fast simulations of a wide range of materials. We demonstrate that more than an order of magnitude performance improvement can be achieved with our GPU solvers. Practical high-resolution examples with up to ten million particles run in less than one minute per frame.

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

Additional Key Words and Phrases: Material Point Method, GPU, SPGrid

ACM Reference Format:
Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods . *ACM Trans. Graph.* 37, 6, Article 254 (November 2018), 12 pages. https://doi.org/10.1145/3272127.3275044

1 INTRODUCTION

The Material Point Method (MPM) is a hybrid Lagrangian/Eulerian computational scheme that has been shown to simulate a large variety of traditionally challenging materials with visually rich animations in computer graphics. Recent examples of MPM-based methods developed for such materials include simulations of snow [Stomakhin et al. 2013], granular solids [Daviet and Bertails-Descoubes 2016; Klár et al. 2016], multi-phase mixtures [Gao et al. 2018a; Stomakhin et al. 2014; Tampubolon et al. 2017], cloth [Jiang et al. 2017a],

*M. Gao, X. Wang, and K. Wu are joint first authors.
†M. Gao was with the University of Wisconsin - Madison during this work.
‡X. Wang was with the Zhejiang University during this work.

Authors' addresses: Ming Gao, University of Pennsylvania; Xinlei Wang, University of Pennsylvania; Kui Wu, University of Utah; Andre Pradhana, University of Pennsylvania , DreamWorks Animation; Eftychios Sifakis, University of Wisconsin - Madison; Cem Yuksel, University of Utah; Chenfanfu Jiang, University of Pennsylvania.

0730-0301/2018/11-ART254 $15.00
https://doi.org/10.1145/3272127.3275044

foam [Yue et al. 2015] and many others. MPM has been shown to be particularly effective for simulations involving a large number of particles with complex interactions. However, the size and the complexity of these simulations lead to substantial demands on computational resources, thereby limiting the practical use cases of MPM in computer graphics applications.

Using the parallel computation power of today's GPUs is an attractive direction for addressing computational requirements of simulations with MPM. However, the algorithmic composition of an MPM simulation pipeline can pose challenges in fully leveraging compute resources in a GPU implementation. Indeed, MPM simulations include multiple stages with different computational profiles, and the choice of data structures and algorithms used for handling some stages can have cascading effects on the performance of the remaining computation. Thus, discovering how to achieve a performant GPU implementation of MPM involves a software-level design-space exploration for determining the favorable combinations of data structures and algorithms for handling each stage.

In this paper we introduce methods for addressing the computational challenges of MPM and extending the capabilities of general simulation systems based on MPM, particularly concentrating on a high-performance GPU implementation. We present a collection of alternative approaches for all components of the MPM simulation on the GPU and provide test results that identify the favorable design choices. We also show that design choices that may superficially appear to be reasonable can suffer from suboptimal performance in practice. Furthermore, we introduce novel methods for thermodynamics and simulation of granular materials with MPM. More specifically, this paper includes the following contributions:

(1) A novel, efficient, and memory-friendly GPU algorithm for accelerated MPM simulation on a GPU-tailored sparse storage variation of CPU SPGrid [Setaluri et al. 2014].
(2) A performance analysis of crucial MPM pipeline components, with several alternative strategies for particle-grid transfers.
(3) A collocated, weak form-consistent, MLS-MPM-based implicit heat solver that enables thermo-mechanical effects on elastoplasticity.
(4) An easy-to-implement unilateral hyperelasticity model with non-associative flow rule for cohesionless granular media.

Our experiments show that more than an order of magnitude performance improvement can be achieved using the favorable choices for data structures and algorithms, as compared to optimized solutions using different computational models. We demonstrate that complex MPM simulations with up to 10 million particles can be simulated within a minute per frame using the methods we describe. We also present results showing that our heat solver can effectively handle phase transition effects and that our hyperelasticity model for granular materials allows achieving consistent simulation results with explicit and implicit integrations.

2 BACKGROUND

2.1 Related Work

Material point method was introduced by Sulsky et al. [1995] as the generalization of the hybrid Fluid Implicit Particle (FLIP) method [Brackbill 1988; Bridson 2008; Zhu and Bridson 2005] to solid mechanics.

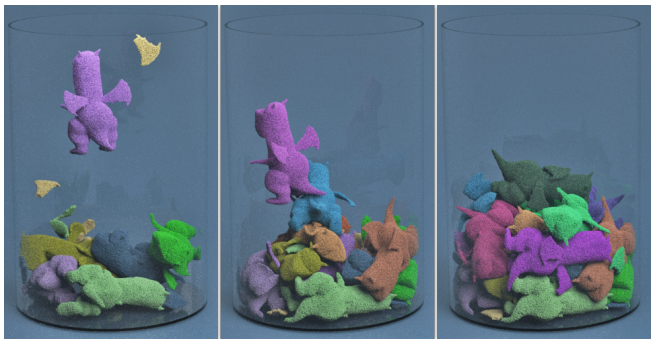

Fig. 2. **How to stack your dragon.** Stacking elastic dragons in a glass. This simulation contains 9.0 million particles on a 512^3 grid with an average 21.8 seconds per $48Hz$ frame.

It has been recognized as a promising discretization choice for animating various solid materials including snow [Stomakhin et al. 2013], foam [Ram et al. 2015; Yue et al. 2015], sand [Daviet and Bertails-Descoubes 2016; Klár et al. 2016], cloth [Guo et al. 2018; Jiang et al. 2017a], fracture [Wretborn et al. 2017], cutting [Hu et al. 2018] and solid fluid mixture [Gao et al. 2018a; Stomakhin et al. 2014; Tampubolon et al. 2017].

For GPU-based simulation methods, many researchers divided the simulation domain in order to parallelize computation on the GPU [Chu et al. 2017; Horvath and Geiger 2009; Liu et al. 2016]. Wang [2018] performed a GPU optimization on sewing pattern adjustment system for cloth. Others explored solutions to the computationally heavy task of self-collision detection, using GPUs [Govindaraju et al. 2007, 2005] with improved spatio-temporal coherence and spatial hashing [Tang et al. 2018, 2013, 2016; Wang et al. 2018; Weller et al. 2017]. In particular, the spatial hashing table [Weller et al. 2017] has been proposed as both an acceleration structure and a substitution to the hierarchy of uniform grids for collision query in order to lower the memory consumption. Furthermore, the histogram sort (alternative to radix sort) has been proven to be capable to significantly reduce the overhead of sorting if the number of bins is adequately smaller than the total count of elements [Wang et al. 2018]. These practices provided great foundations for our MPM pipeline as they fit well with the sparse grid structure.

From the Eulerian view, the simulation domain is represented by a discretized grid. Museth [2013] developed OpenVDB, which is a tree with a high branching factor that yields a large uniform grid at leaf nodes. This adaptive data structure has been shown to be very efficient and broadly used. Inspired by that, Hoetzlein et al. [2016] proposed GVDB Voxels, a GPU sparse grid structure. The voxel data is represented in dense n^3 bricks allocated from a pool of sub-volumes in a *voxel atlas*. The atlas is implemented as a 3D hardware texture to enable trilinear interpolation and GPU texture cache. Recently, Wu et al. [2018] extended it with dynamic topology update and GPU optimized matrix-free conjugate gradient solver for fluid simulations with tens of millions particles. Although the use of texture to store volumetric data can benefit performance for general purpose usage, such as hardware trilinear interpolation and fast data accessing, it prevents GVDB from using scattering rather than gathering because atomic operations on textures are not allowed to be used under current GPU hardware.

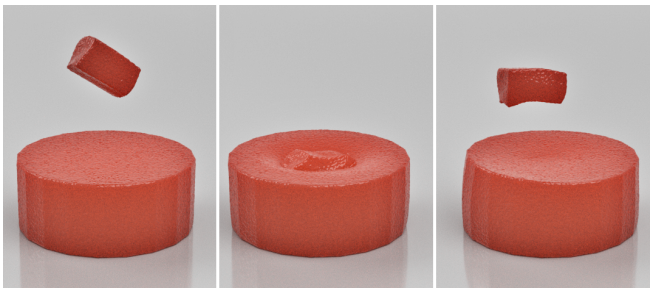

Fig. 3. Elasticity simulation of Gelatin bouncing off Gelatin with 6.9 million particles on a 512^3 grid at an average 6.72 seconds per $48Hz$ frame.

SPGrid [Gao 2018; Setaluri et al. 2014] provided an alternative sparse data structure; it has been adopted in large-scale fluid simulations [Aanjaneya et al. 2017; Liu et al. 2016; Setaluri et al. 2014] and in the MPM context [Gao et al. 2018a, 2017; Hu et al. 2018]. SPGrid improves the data locality by mapping from a sparse 2D/3D array to a linear memory span by following a modified Morton coding map. Furthermore, it exploits hardware functions to accelerate the translation between geometric indices and the 64-bit memory offsets. The neighborhood accesses can be achieved in $O(1)$, rather than $O(\log n)$ for traditional tree-based sparse storage schemes (n is the number of the leaf nodes).

2.2 MPM Overview

MPM, as a hybrid spatial discretization method, benefits from the advantages of both Lagrangian and Eulerian views. MPM uses Lagrangian particles to carry material states including mass m_p, position $\mathbf{x}_p$, velocity $\mathbf{v}_p$, volume V_p, deformation gradient $\mathbf{F}_p$, etc. The grid acts as an Eulerian scratchpad for computing stress divergence and performing numerical integration. Grid nodes represent the actual degrees of freedom, which store mass m_i, position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$ on each node i.

A typical first-order MPM time integration scheme for incremental dynamics from t^0 to t^1 (with $\Delta t = t^1 - t^0$) contains the following essential steps:

(1) Particles-to-grid (P2G) transfer of masses and velocities: $\{m_i^0, \mathbf{v}_i^0\} \leftarrow \{m_p, \mathbf{v}_p^0\}$;
(2) Grid velocity update using either explicit or implicit integration: $\mathbf{v}_i^1 \leftarrow \mathbf{v}_i^0$;
(3) Grid-to-particles (G2P) transfer of velocities and strain increments: $\{\mathbf{v}_p^1, \mathbf{F}_p^1\} \leftarrow \mathbf{v}_i^1$;
(4) Particle-wise stress evaluation and plasticity treatment that modifies $\mathbf{F}_p^1$.

Assuming the usage of a matrix-free Krylov solver for the linearized system (due to its superior efficiency in implicit MPM, where dynamically changing sparsity pattern of the stiffness matrix could cause significant overhead in explicitly storing the corresponding linear system), both explicit and implicit Euler time integrations of MPM break the entire computation procedures in (1)-(3) into particle-grid transfer operations of physical quantities and their differentials. As such, the key to high-performance MPM is the optimization of particle-grid transfer operators.

Unsurprisingly, the algorithmic choice of the transfer scheme plays a considerable part in affecting the computing performance. We design our benchmarks based on three possible choices of the transfer scheme: FLIP [Zhu and Bridson 2005], APIC [Jiang et al. 2015] and the recently proposed MLS-MPM [Hu et al. 2018]. Note that MLS-MPM provides an additional algorithmic speed-up by avoiding kernel weight gradient computations in step (2) (discussed in more detail in §4 and by Hu et al. [Hu et al. 2018]).

2.3 Particle-to-Grid Transfer

As in many other Lagrangian-Eulerian hybrid methods, the particles carry material states and transfer them to the grid as mentioned in §2.2 step (1). On the GPU, the two most common approaches for performing parallel state transfer are *gathering* and *scattering*. The former one gathers states from all nearby particles to one particular grid node; while the latter one distributes all states of one particular particle to all influenced grid nodes.

The domain decomposition technique is commonly used in the gathering strategy [Huang et al. 2008; Parker 2006; Stantchev et al. 2008; Zhang et al. 2010]. Using this strategy the simulation domain is divided into several small sub-domains such that each node only needs to check all the particles within the neighboring sub-domains instead of the whole domain. Binning is used for avoiding conflicts in atomic operations and accessing each particle only once [Klár 2014, 2017; Klár et al. 2017], but the precomputation can be very expensive and having a different number of particles per cell leads to thread divergence. Ghost particles are introduced to minimize the number of communications and barriers within sub-domains [Parker 2006; Ruggirello and Schumacher 2014]. Chiang et al. [2009] maintain and update particle lists for each node during the simulation. Obviously, it requires a huge amount of GPU memory as well as the processing time for updating particle lists. In order to reduce the workload, Dong et al. [2015] extend the influencing range of the grid nodes and further adapt their method to multiple GPUs [Dong and Grabe 2018]. Wu et al. [2018] pre-compute a subset of the particles for each grid cell. However, their method still needs to unnecessarily examine a large amount of particles within each voxel. Furthermore, for storing the subsets, it not only requires expensive data movements but also consumes a large amount of memory. To summarize, all the fundamental issues of gathering are due to the need for accessing associated particles of a node. There is no satisfying solution reading the data efficiently as well as saving the extra memory for particle lists, etc. The other inherent problem is that the workload of each GPU thread is closely related to the length of the particle list of each grid node, thus thread divergence generally exists and slows down P2G.

On the contrary, the scattering method is free from all these critical issues, but its performance suffers from write conflicts. Once the write conflicts are resolved, the scatter-based transfer scheme should be superior, because it has more balanced workload and it also avoids the overhead of maintaining particle lists. Both Harada et al. [2007] and Zhang et al. [2008] introduced parallel scattering methods to distribute particle attributes to neighboring particles in SPH. In their methods, particle attributes are written into 2D texture buffer simultaneously using graphic rasterization pipeline to solve write-conflicts. However, their scattering methods cannot be easily employed in a 3D Eulerian simulation framework because the current hardware rasterization technique only works for 2D texture,

Algorithm 1 Sparse MPM simulation on the GPU

```
1:  procedure SparseMPM( )
2:      P ← initial points
3:      P ← sort(P)                                   ▷ Section 3.3
4:      for each timestep do
5:          dt ← compute_dt(P)
6:          V ← GSPGrid structure(P)                  ▷ Section 3.1
7:          M ← particle_grid_mapping(P, V)           ▷ Section 3.1
8:          V ← particles_to_grid(P, M)               ▷ Section 3.2
9:          V ← apply_external_force(V)
10:         V ← grid_solve(V, dt)
11:         P ← grid_to_particles(V, M)
12:         P ← update_positions(P, dt)
13:         P ← sort(P)                               ▷ Section 3.3
```

not to mention the added complexity to build a mapping between a sparse grid and 2D texture during rasterization. Fang et al. [2018] and Hu et al. [2018] recently proposed an asynchronous MPM simulation method for acceleration on the CPU. Their implementation adopted the parallel scatter-based scheme and thus also suffered from the data race during the particle-to-grid transfer. To avoid the expensive per-cell locking and guarantee correctness as well, they partitioned all the blocks into several sets, in each of which no two blocks share any overlapping grid node. But this method only prevented the write conflicts between blocks, since blocks were used as the scheduling units on the CPU and it cannot resolve the conflicts when multiple particles are writing to the same grid node simultaneously, which is the case on the GPU. We propose a novel scatter-based GPU implementation of transfer kernels, which alleviates the heavy use of atomic operations by utilizing modern graphics hardware features, and thus avoids the majority of the overhead.

3 OPTIMIZED GPU SCHEME FOR MPM

In this section we describe our optimized GPU scheme for MPM using a GPU-tailored sparse paged grid data structure. The overall algorithm is summarized in Algorithm 1.

3.1 GSPGrid Tailored to MPM

We introduce *GSPGrid*, the GPU adaptation of the SPGrid [Setaluri et al. 2014] data structure which facilitates the sparse storage required for efficient simulations. Though not limited to such a choice, a $4 \times 4 \times 4$ spatial dissection of the computational domain into GSPGrid blocks is assumed in our implementation.

We take a similar strategy to [Gao et al. 2017], and briefly summarize it here. We use the quadratic B-spline functions as the weighting kernel, so each particle is associated with $3\times3\times3$ grid nodes (3×3 in 2D as shown in the Fig. 4). We assign each particle to the cell whose "min" corner collocates with the "smallest" node in the particle's local $3\times3\times3$ grid. All particles whose corresponding cells are within the same GSPGrid block are attached to that block. Geometrically, the particles of a particular GSPGrid block all reside in the same dual block, i.e. the $\Delta x/2$-shifted block (the red dashed block in Fig. 4). All computations for transferring particles' properties to/from the grid will be conducted locally in the same GSPGrid block.

We assign each particle to one CUDA thread. The particles are sorted such that the computations of particles sharing the same dual cell are always conducted by consecutive threads. In scenarios involving high particle densities, the number of particles within a single GSPGrid block can easily go beyond the maximum number of threads allowed in a CUDA block under the current graphic architecture. To solve this issue, we assign each GSPGrid block to one or several CUDA blocks and generate the corresponding *virtual-to-physical* page mapping. In this way, we can treat each CUDA block separately without considering their geometric connections.

Notice that, for some particles, e.g. the ones in the yellow cell in Fig. 4, reading and writing will involve nodes from neighboring blocks. To deal with such particles, the shared memory of each CUDA block temporarily allocates enough space for all 7 neighboring blocks (3 in 2D). One CUDA block usually handles hundreds of particles in parallel, which makes it affordable to allocate enough shared memory for all neighboring blocks. For example, in grid-to-particle transfer, we can first fetch all required data into the shared memory from the global grid, and then update particles' properties.

As discussed above, it is possible for a particular block to access data from its neighboring blocks. In SPGrid, the offsets of the neighboring blocks can be easily computed for addressing them. However, without the support of virtual memory space in GPU, we also need to store the address information of neighboring blocks. Fortunately, the spatial hashing algorithm is able to construct the topology of neighboring blocks in $O(1)$ complexity on the GPU.

3.2 Parallel Particle-to-Grid Scattering

For particle-to-grid transfers, geometrically neighboring particles can write into the same nodes, making write hazards a critical problem which has been examined in many Eulerian-Lagrangian hybrid methods as discussed in §2.3. There are generally two schemes for resolving this problem, scattering and gathering. Scattering methods simply use atomic operations to avoid conflicts. On the other hand, for gathering, usually a list of particles are created and maintained during the simulation; thus each node can track down all the particles within its affecting range.

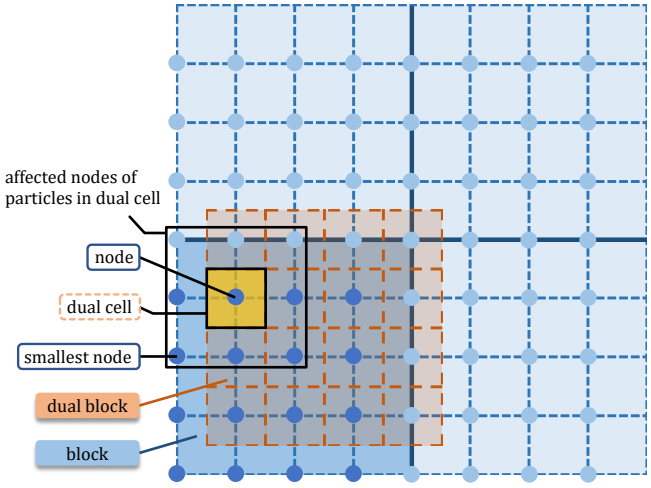

Fig. 4. **Mapping from particles to blocks:** All particles in the yellow dual cell interact with the same set of 27 nodes (9 in 2D). These particles also distribute states to the grid nodes of the top neighboring block.

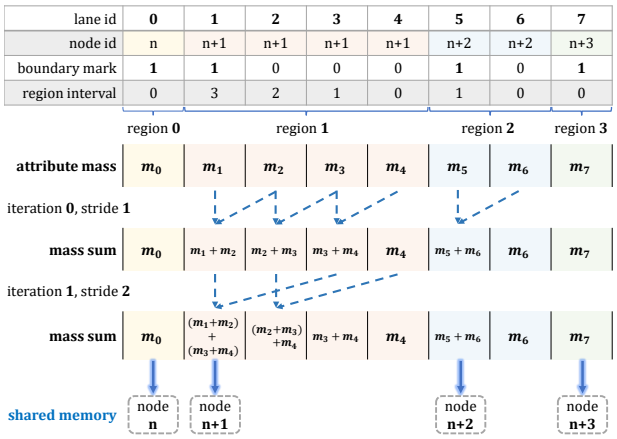

Fig. 5. **Optimized particles-to-grid transfer**. By using CUDA warp intrinsics to accelerate attribute summations, results are first written to the shared memory and then transfered to global memory in bulk.

Almost all previous papers opt for a gathering approach since it is widely believed that highly frequent atomic operations in scattering can significantly undermine the performance especially in GPU-based parallel applications. However, we propose a method to handle most of the write conflicts from scattering without atomic operations, inspired by the concept of parallel reduction sum [Luitjens 2014] and show that the performance is superior to the gathering ones. As we divide the whole grid domain as well as their corresponding particles into blocks and cells, there are two levels of write hazards in P2G, within-block hazards and crossing-block hazards. As observed in practice, the within-block conflicts are the absolute majority. The within-block ones can be further divided into within-warp and crossing-warp conflicts and our solution particularly tackles the within-warp ones.

As shown in Fig. 5, within a warp, a few groups of particles tend to add their attributes to the corresponding grid nodes. For particles of a particular group (e.g., particles 1-4), simultaneous write operations into the same location are conventionally done through atomic operations. We exploit the warp-level CUDA intrinsic functions, i.e. *ballot* and *shfl*, to resolve this problem. First, a representative of each group is chosen (i.e. the left most one whose boundary mark is set on). Then attributes from all the other particles within the same group are added to the one from the representative by iteratively shuffling from right to left. The stride of shuffle doubles after each iteration, and the total number of iterations should be enough to cover the longest region in the warp by using Algorithm 2. Finally, the representative particle is responsible for writing the sum to the target grid node, as illustrated in Algorithm 3. In this way, all warp-scale write conflicts are eliminated. To further reduce the global write conflicts across different warps, the shared memory acts as a buffer for temporarily holding the summation results from different warps. Note that this crossing-warp conflicts only infrequently happen such that the negative impact caused by atomic-adds to the shared memory is almost negligible.

Algorithm 2 Warp Computation

```
1:  procedure ComputeBoundaryAndInterval(int laneid, int* cellids)
2:      cellid ← cellids[laneid]
3:      cellid_prev ← cellids[laneid − 1]
4:      if laneid = 0 ∨ cellid ≠ cellid_prev then
5:          boundary ← true
6:      mark ← brev(ballot(boundary))
7:      interval ← countFollowingZeroes(mark, laneid)
8:      stride ← 1
9:      iter ← interval
10:     while stride < 32 do
11:         tmp ← shfldown(iter, stride)
12:         iter ← max(tmp, iter)
13:         stride ← stride << 1              ▷ //move on to a higher level
14:     iter ← shfl(iter, 0)                  ▷ //broadcast the maximum iterations
```

3.3 Cell-based Particle Sorting

Since the positions of the particles are updated in each time-step, the GSPGrid data structure should be refreshed accordingly. To build the new mapping from the GSPGrid blocks to the continuous GPU memory, we need first to re-sort the particles to help identify all the blocks occupied or touched by the particles.

SPGrid translates the cell index of each particle to a 64-bit integer offset, which is used as the keyword for sorting. Radix sort is generally considered the fastest sorting algorithm on GPU. In our simulations, the count of offsets in use is almost negligible compared to what a 64-bit integer can represent. We therefore use spatial hashing to transform these sparse offsets of GSPGrid blocks into consecutive numbers. Whenever a new block is touched, the index assigned to the block increments by one (starting from 0). The transformed block indices of all the particles are thus consecutive, and the maximum number is usually several orders of magnitude smaller than the particle number. Furthermore, all particles within the same block can be partitioned by $4 \times 4 \times 4$ cells. This particular layout is required by the follow-up computations including precalculating the smallest index of each particle and reducing write conflicts during P2G transfer (§3.2).

With CPU SPGrid, radix sort facilitates proximity (in memory) of geometrically neighboring blocks to improve prefetching efficiency. In GPU, this is not a concern anymore; thus we use histogram sort instead of the conventional radix sort. The new keyword, which is a combination of the transformed block index and dense cell index within the block, works as the reference to the bin. The observed performance speedup approaches an order of magnitude, and in collaboration with *delayed ordering technique* (§3.4), reordering the particles is no longer a bottleneck.

Algorithm 3 Warp Write

```
1:  procedure GatherAndWrite(T* buffer, T attrib, int iter)
2:      stride ← 1
3:      val ← attrib
4:      while stride <= iter do                ▷ //hierarchical summation
5:          tmp ← shfldown(val, stride)
6:          if stride <= interval then         ▷ //only sum within the group
7:              val ← val + tmp
8:          stride ← stride << 1               ▷ //move on to a higher level
9:      if boundary then                       ▷ //only the boundary node needs to write
10:         *buffer ← *buffer + val            ▷ //AtomicAdd is applied
```

3.4 Particle Reordering

Given the sorted indices from §3.3, one natural thing to do next is to reorder all particles' properties accordingly since coalesced memory accesses are always preferable in CUDA kernels. However, for a high-resolution simulation, the reordering itself can be the bottleneck, since each particle carries various properties such as positions, velocities, and deformation gradient, etc., and reading and writing those data in a non-coalesced manner can be time consuming. We propose a practical way to completely dispense with the reordering. We observe that any functions which take in scattered data and write back the updated results in order can actually sort the data as a byproduct. In essence, a reordering function simply writes unordered inputs back in a different order. If we apply the reordering function to the particle property optionally rather than sorting all the properties of a particle, the cost / latency due to the scattered memory reads can be largely mitigated and the overhead of reordering is avoided.

The position of a particle is the only property we decide to reorder since they are essential for almost all kernels that build mappings between particles and grid blocks / cells, and for all transfer kernels that needs to compute weights and weight gradients from the positions. We treat all the other properties accordingly based on how they are used in the related computations. We list a few of them as the most typical cases. Note that most CUDA kernels follow the positions' order, except for certain material-based computations.

Mass. Since each particle's mass remains constant, their sequence will never be changed; for retrieving the correct mass, we only need to map from the current particles' order to the original order.

Velocity. Velocities are updated every time-step in the G2P kernel. The new velocities are forced to follow the order of the threads / positions by simply writing them back in a coalesced manner. However, when the velocities are being used as the inputs to some kernels in the next time-step, they are not matching with new positions' order since we choose only to reorder positions at the end of each time-step. As a result, a mapping from the current order to the previous order is required.

Stress. When computing the stresses, we choose not to change the order of either the inputs $\mathbf{F}$ or the outputs $\mathbf{P}$. One reason is that the GPU SVD function can be optimized (§4.3) so fast that it may be inefficient to amortize the scattered writes.

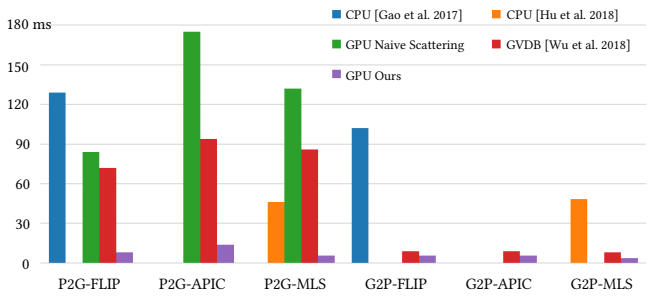

Fig. 6. **Transfer benchmark.** Comparison of our GPU scattering to a SIMD CPU implementations of FLIP [Gao et al. 2017] and MLS [Hu et al. 2018] transfer schemes, a naive GPU scattering implementation using atomic operations, and a GPU gathering implementation using GVDB [Wu et al. 2018] on Nvidia TITAN Xp.

4 BENCHMARKS AND PERFORMANCE EVALUATION

To evaluate our GPU MPM algorithm, we create several benchmarks, starting with the uniform particle distribution (§4.1). Firstly, we compare the performance of the transfer kernels between our method and one SIMD-optimized multi-core CPU implementation as well as one gathering-based GPU implementation with GVDB as the sparse background grid. Secondly, with the total number of particles being fixed, we vary the particle densities (particles per cell, PPC) to examine how our pipeline can be affected.

We further evaluate the performance with non-uniform particle distributions, e.g. Gaussian distributions (§4.2). In this experiment, we fix both the total number of particles and the total number of cells being occupied by the particles; while the PPC of each single cell can vary following particular Gaussian distributions.

In addition to the explicit pipeline, we also measure the performance of the key kernels merely used in the implicit time integration [Stomakhin et al. 2013]. Since the results of the singular value decomposition (SVD) of the deformation gradient are required in the computation of the stress derivative in every single iteration, one can always pre-compute the SVD and store the results in advance. However, our experiment (§4.3) reveals some interesting findings.

Notice that, as mentioned before, the three-dimensional GSPGrid block with 16 channels is of resolution $4 \times 4 \times 4$ and eight particles per cell are usually required for stability considerations in MPM applications. Hence, we choose to allow each CUDA block to process at most 512 particles, due to the limitation of the current hardware architecture.

Unless otherwise stated, all of our GPU tests are performed on Quadro P6000 and all CPU tests are performed on an 18-core Intel Xeon Gold 6140 CPU.

4.1 Uniform Distribution Benchmarks

4.1.1 Comparisons with two state-of-the-art implementations. We create a benchmark with $\Delta x = \frac{1}{128}$. The particles are uniformly sampled on a grid from $(\frac{1}{8}, \frac{1}{8}, \frac{1}{8})$ to $(\frac{7}{8}, \frac{7}{8}, \frac{7}{8})$ with spacing $\frac{1}{256}$. The total number of the particles is just over 7 million particles. For this test, the CPU benchmark was performed on an 18-core Intel Xeon Gold 6140 CPU and all GPU measurements were performed on a NVIDIA TITAN Xp. The results are in Fig. 6.

Comparison with CPU implementation. We compare our scatter-based GPU implementation of particle-grid transfers to SIMD optimized CPU implementations of FLIP [Gao et al. 2017] and MLS [Hu et al. 2018]. In our tests, our P2G and G2P kernels achieved more than 16× speedups in comparison to the CPU implementation of FLIP [Gao et al. 2017], and about 8× and 13× speedups as compared to the CPU implementation of MLS [Hu et al. 2018], respectively.

Comparison with GPU implementation with atomic operations. We perform comparisons against an atomic-only scattering implementation, which is an order of magnitude slower than our proposed method as shown in Fig. 6. Of course, the fact that we did not find this route to be attractively efficient in current platforms, is no

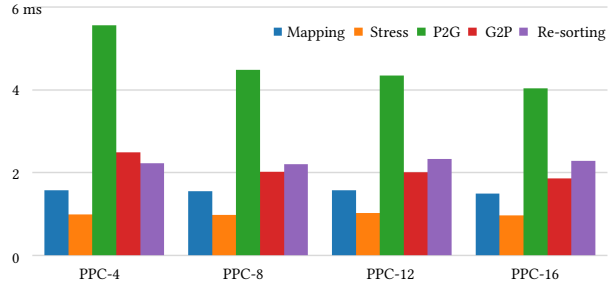

Fig. 7. **Particle density benchmark.** The total number of particles is approximately fixed as 3.5M. The stress kernel includes the SVD computation.

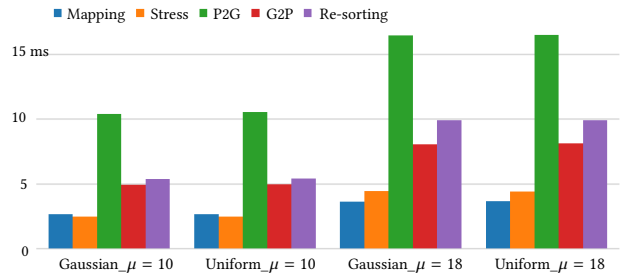

Fig. 8. **Gaussian benchmark.** We compare the performance of each critical kernel when the particle-per-cell distributions following Gaussian and uniform distributions The stress kernel also includes the SVD computation.

indication that the performance of atomics will not be faster in future generations. However, since MPM for simulating solids needs quadratic kernels to acquire continuous force fields, the amount of conflicts encountered is usually much more than expected.

Comparison with GVDB-based implementation. We compare with one of the most recent GPU gathering implementation with sparse grid structure [Wu et al. 2018]. Their idea is basically to pre-compute a subset of particles for each grid cell. All particles influencing the grid nodes inside the cell will be included in that list. When performing the particle-to-grid transfer, each node needs to check all particles in the list to determine whether they are close enough. Therefore, each node has to check much more unnecessary particles than needed (only 20% utilization for MPM FLIP with subcell size 4^3). In order to do a comprehensive comparison, we use three different transfer schemes, including FLIP, APIC, and MLS. Since their gathering method needs to create lists for all particle attributes, such as velocity, position, stress, and deformation gradients, it takes half of the computation time to load and store data. More importantly, it consumes tremendous amounts of GPU memory.

For our P2G kernels, the computing workload and the memory access workload are well balanced. Therefore, the timing is bounded by both memory and computations. FLIP only needs to compute nodal mass, traditional translational momentum and forces; while APIC also needs to load one additional matrix for the affine velocity modes. In contrast, MLS completely avoids the computation of weight gradients, and the two matrix-vector multiplications of APIC (i.e. computing the force and the affine modes) can be merged into one [Hu et al. 2018].

For our G2P kernels, memory is utilized more heavily than computing units. Compared to APIC and MLS, FLIP also needs to load the nodal velocity increments for updating the particles' velocities. However, APIC and MLS have to refresh one extra matrix for recording the affine velocity modes; while MLS can merge the updates of $\mathbf{F}$ and that extra matrix into one to reduce the total cost [Hu et al. 2018]. GVDB uses a 3D texture to store volume data to utilize the hardware trilinear texture interpolation functions; however MPM cannot benefit from this because of the higher order B-spline weighting functions. Furthermore, we exploit the shared memory to pre-load grid data for all particles within the same block.

4.1.2 Particle density benchmark. In this subsection, while fixing the total number of particles, we run the benchmarks for cases with different particle densities (particles per cell, PPC). And we start to include all other critical kernels in addition to the transfer kernels; all tests are conducted using the MLS transfer scheme. As shown in Fig. 7, it is reasonable to observe that when the particle density increases, the transfer kernels take less time to finish, since the higher PPC renders a smaller sparse grid structure. For the other kernels, which are mostly particle-oriented, i.e. the underlying grid structure does not really interfere with them, the impacts of the varying PPC seem to be negligible.

4.2 Gaussian Particle Distribution Benchmarks

To further examine the impacts of non-uniform particle distributions, we also run some benchmarks in which the particle-per-cell varies based on a Gaussian distribution. All tests are with MLS. We use the same box domain from $(\frac{1}{8}, \frac{1}{8}, \frac{1}{8})$ to $(\frac{7}{8}, \frac{7}{8}, \frac{7}{8})$. For the two Gaussian distributions, the minimum particle-per-cell are 4 while the maximums are 16 and 32; while the corresponding uniform cases are with particle-per-cell being 10 and 18 respectively. As shown in Fig. 8, the performances are almost identical, proving that our scheme is not affected by the particle distribution when the background sparse grid remains the same.

4.3 Implicit Iteration and SVD

We adopt the matrix-free Krylov solver for the implicit step in which the multiplication of the system matrix and a vector can be expressed by concatenating a G2P transfer and a P2G transfer (ref.[Stomakhin et al. 2013] for more details). Notice those two transfer kernels are not the same as the ones used in the explicit MPM solver. We name them as P2G-Implicit and G2P-Implicit kernels as in Fig. 9. Both FLIP and APIC (non-MLS) have to compute weight gradients while MLS approximates weight gradients with weights. From the right half of Fig. 9, the G2P-MLS is slightly slower than G2P-non-MLS because G2P-MLS needs to do additional 9 multiplications due to the extra term $(\mathbf{x}_p - \mathbf{x}_i)$ at all 27 nodes.

We also consider the possibility of precomputing and storing the results of SVD at the beginning of each time step. Whenever the stress kernel or stress-derivative kernel needs, we simply load the SVD results from memory. Notice that, for stress kernel, it is faster to simply re-compute SVD repeatedly; while for stress-derivative kernel, there is only negligible difference. The main reason for this discrepant behavior in the two kernels is that the computing workload in Stress kernel is already lighter than the memory workload, loading more data in can significantly impede the performance. On the other hand, Stress-derivative has enough computing workload to mitigate the memory cost for loading SVD results.

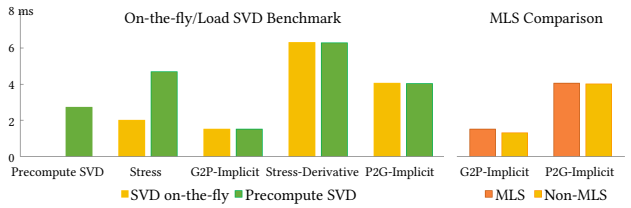

Fig. 9. **On-the-fly/load SVD benchmark and MLS comparison.** Left, when we pre-compute SVD and store the results, the stress and stress-derivative kernels load the SVD results directly from the memory; otherwise they recompute SVD on-the-fly ; right, we compare the performance for one implicit iteration of MLS and non-MLS implicit integrations.

In the same CPU used in Sec. 4.1.1, a AVX512 SVD implementation of [McAdams et al. 2011] takes about 2.3 ns per particle (SVD) and implicit symmetric QR SVD [Gast et al. 2016] takes 17.0 ns; while our GPU implementation takes about 0.37 ns.

4.4 Reorder Benchmark

We compare the performances of 7M particles example with particle reordering and without particle reordering (i.e. delayed reordering) using the explicit integration. As shown in Fig. 10, computing stress, P2G, and G2P are barely affected by reordering, while our method only reorders the particle positions rather than all attributes such as velocity and deformation gradient.

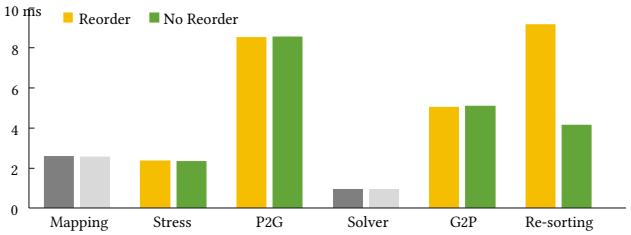

Fig. 10. **Reorder benchmark.** Our delayed ordering technique can reduce sorting time dramatically while all other kernels are barely impacted. Only colored components are impacted by reordering.

4.5 Integration Benchmark

We compare our explicit integration pipeline with the matrix-free implicit Krylov solver by simulating the collision between two elastic dragons and list the performance in (Fig. 11). The Δt of them are set to be 1×10^{-4} and 1×10^{-3} respectively. Obviously, the implicit solver spends more time per step to solve a linear system, but also can use larger time step. The source code of this benchmark is included in the supplemental materials.

5 MPM HEAT SOLVER WITH MLS SHAPE FUNCTIONS

MPM can be generalized to derive an implicit scheme for solving the heat equation. We follow Hu et al. [2018] in deriving a moving least squares (MLS) weak form. The resulting algorithm allows us to accurately capture heat conduction in virtual materials and enables us to simulate thermo-mechanical phenomena such as melting.

Stomakhin et al. [2014] also investigated thermo-mechanical effects in the context of MPM. Their formulation is based on a

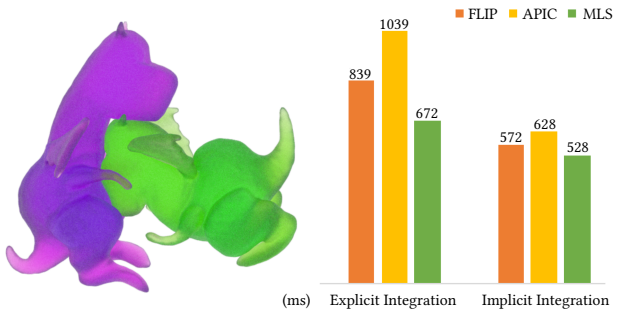

Fig. 11. **Integration benchmark.** We compare the performance of the two dragons colliding example (left) with our GPU explicit and implicit solver along with three different schemes (right). The total number of particles is 775K and the grid resolution is 256^3.

staggered-grid finite difference discretization which requires heuristic boundary cell labeling. Our method, in contrast, naturally enforces the zero Neumann boundary condition (insulated at the free surface) when no surface heat flux is specified. This boundary condition is analogous to the zero traction boundary condition in discretizing the momentum equation with MPM. On the other hand, Dirichlet boundary temperatures are enforced at nodes by prescribing the values. Performing a weak form consistent discretization allows us to treat particles as mass-full quadrature points. Consequently, unlike Stomakhin et al. [2014], we do not need to transfer particle-wise heat capacity or conductivity to the grid. We also maintain a consistent discretization function space for both the momentum and heat equations.

Additionally, our method shares the same collocated MPM grid and transfer kernels as the ones used for solving the momentum equation. The existing optimization strategies for velocity and force transfers on the GPUs directly apply to temperature and "thermal force" transfers with negligible modifications.

5.1 Continuous Equation

We start from the Eulerian-form heat equation

$$\rho(\mathbf{x},t)\,c(\mathbf{x},t)\,\frac{D\theta(\mathbf{x},t)}{Dt} - \nabla\cdot\kappa(\mathbf{x},t)\,\nabla\theta + q^{ext} = 0,$$

where $\mathbf{x}$ and t are the current position and time, θ is temperature, ρ is density, c is Eulerian specific heat capacity (with its Lagrangian counter part $C(\mathbf{X},t)$ and unit $J/(kg\cdot K)$, where $\mathbf{X}$ is the reference position), κ is heat conductivity, q^{ext} encodes any external body heat source such as radiation, and

$$\frac{D\theta(\mathbf{x},t)}{Dt} = \frac{\partial\theta}{\partial t} + \mathbf{v}(\mathbf{x},t)\cdot\nabla\theta(\mathbf{x},t)$$

is the material derivative of θ.

5.2 MLS Discretization

In deriving the weak form of the heat equation, we follow the discretization strategy of Hu et al. [2018] closely. The backward Euler discretization (from t^0 to t^1) of the weak form of the heat equation is given by

$$\frac{\hat{\mathcal{M}}_i^0(\theta_i^1-\theta_i^0)}{\Delta t} = \mathbf{q}^1 + \mathbf{y}, \tag{1}$$

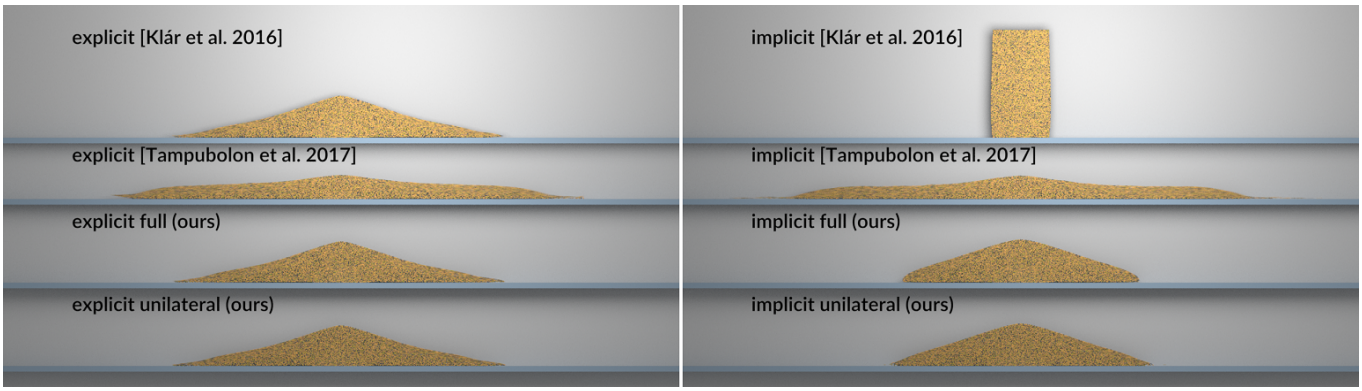

Fig. 12. **Sand constitutive model.** The left and right columns depict explicit and semi-implicit simulations respectively. The semi-implicit scheme with the St. Venant-Kirchhoff constitutive model used in [Klár et al. 2016] introduced a severe numerical viscosity, while the modification proposed by [Tampubolon et al. 2017] introduced spreading effect and non-physical column-collapse profile. Our proposed energy density functions mitigate these shortcomings.

where $\mathbf{q}^1(\mathbf{x}) = \int_{\partial\Omega^0} \left(\Phi_i \kappa^0(\mathbf{x}) \theta_j^1 \nabla\Phi_j\right) \cdot \mathbf{n}\, ds(\mathbf{x})$ encodes the heat flux Neumann boundary condition,

$$\mathbf{y}(\theta^1) = -\int_{\Omega^0} \nabla\Phi_i \cdot \left(\kappa^0(\mathbf{x}) \theta_j^1 \nabla\Phi_j\right) d\mathbf{x}$$

is the implicit "thermal force". $\hat{\mathcal{M}}_i^0 = \sum_p m_p C_p N_i(\mathbf{x}_p)$ is the lumped thermal mass, with $N_i(\mathbf{x})$ being the quadratic B-spline interpolation function. MLS shape functions $\Phi_i(\mathbf{x})$ are used to reconstruct a function space near each particle. The integration domain is further expressed as a summation over particle domains, where each integral over a particle domain is approximated using one point quadrature. The resulting formulation for an insulated body with no boundary heat flux is then

$$\frac{1}{\Delta t}(\theta_i^1 - \theta_i^0)\hat{\mathcal{M}}_i^0 = -\sum_j \left(\sum_p \kappa_p V_p^0 \nabla\Phi_i(\mathbf{x}_p) \cdot \nabla\Phi_j(\mathbf{x}_p)\right) \theta_j^1, \quad (2)$$

where θ_i is temperature of node i, C_p is the specific heat capacity of particle p, and κ_p is the heat conductivity of particle p.

Utilizing the MLS shape functions avoids differentiating B-spline kernels in high dimensions. More specifically, if a linear polynomial space with quadratic B-spline weighting is chosen for the MLS reconstruction, we have [Hu et al. 2018]

$$\nabla\Phi_i(\mathbf{x}_p) = D_p^{-1} N_i(\mathbf{x}_p^0)(\mathbf{x}_i - \mathbf{x}_p^0),$$

where $D_p = \frac{1}{4}\Delta x^2$ for the quadratic B-spline function.

After the temperature increment is solved on the grid, we transfer it back to the particles during the grid-to-particles step. A more detailed explanation of the discretization step is provided in [Gao et al. 2018b].

6 CONSTITUTIVE MODELS

6.1 Temperature Dependent Elasticity

The elastic response of the simulated material is modeled in the isotropic hyperelasticity framework. In this context, the energy density function is a function of the singular values of the deformation gradient whose SVD is given by $\mathbf{F} = \mathbf{U}\hat{\mathbf{F}}\mathbf{V}^T$, with $\hat{\mathbf{F}} = \text{diag}\{\hat{f}_0, \hat{f}_1, \hat{f}_2\}$. For temperature dependent physical model, we adopt the fixed corotated energy density function as proposed in [Stomakhin et al. 2012]:

$$\hat{\psi}(\hat{\mathbf{F}}) = \mu(\theta) \sum_{i=0}^{d-1} (\hat{f}_i - 1)^2 + \frac{\lambda(\theta)}{2} (\det(\hat{\mathbf{F}}) - 1)^2. \quad (3)$$

Here, d represents the number of spatial dimensions. We use interpolated grid temperature values at particle locations instead of θ_p on particles to drive the phase change. This prevents the influence of the ringing instability [Jiang et al. 2015, 2017b], i.e., particle temperature modes that are invisible to the integrator due to the null space in the particles-to-grid transfer operator.

We additionally apply a numerical RPIC damping [Gao et al. 2018a; Jiang et al. 2017a] to achieve the look of viscous flow in the fluid phase.

6.2 Stabilizing Shear Compliant Particles

When a phase change occurs, the shear modulus μ is set to 0. In this case the energy density only penalizes volumetric change without penalizing shearing. This process has been shown to cause the entries of $\mathbf{F}$ to grow unbounded quickly (even close to the square root of `FLT_MAX`) while its determinant stays close to 1. Floating point accuracy is correspondingly drastically affected and floating point overflow can easily get triggered. One solution is to use an equation of states constitutive model which only depends on the update of the determinant of $\mathbf{F}$ as in [Tampubolon et al. 2017]. Instead, we adopt a simple solution as in [Stomakhin et al. 2014] by projecting $\mathbf{F}$ to the hydrostatic axis and setting the diagonal entries of it to be $J^{1/d}$. This corresponds to a plasticity return mapping that absorbs the isochoric part of the elastic deformation gradient into the plastic part while only keeping its dilational part. This strategy improves the numerical stability of our algorithm significantly.

6.3 Unilateral Model for Cohesionless Granular Material

We simulate cohesionless sand as an elastoplastic material. We propose a *full* quartic model whose energy density function is given

Table 1. **Average simulation time per frame.** All timings are in seconds and frame rate is 48.

	Particles #	Domain	Δt	Mapping	Stress	P2G	Solver*	Solver w/ Heat*	G2P	Sorting	Others	Total (s)
Dragon Cup	9.0M	512^3	1×10^{-4}	0.64	0.57	2.30	13.94	-	1.00	1.35	1.15	**20.95**
Granulation	6.7M	512^3	2.5×10^{-4}	0.26	1.34	1.88	33.47	-	0.74	0.41	0.32	**38.42**
Gelatin	6.9M	512^3	1×10^{-3}	0.08	0.05	0.26	5.50	-	0.10	0.26	0.02	**6.27**
Melting	4.2M	256^3	8×10^{-3}	0.02	0.01	0.07	-	10.32	0.02	0.03	0.01	**10.48**

** Implicit solvers were used for all tests in this table. The time for **P2G** and **G2P** used in the solver is not included in this table. Details refer to Table 2.*

Table 2. **Average percentages for all components in the solver.** The MLS transfer scheme is used for the implicit solver.

	Compute			
	G2P	Gradient & SVD	P2G	Others
Dragon Cup	12.8%	36.8%	43.3%	7.1%
Granulation	13.4%	12.9%	45.3%	28.4%
Gelatin	12.4%	34.2%	42.9%	10.6%
Melting	11.3%	37.0%	39.5%	12.2%

by

$$\hat{\psi}(\hat{\mathbf{F}}) = a\mu \sum_{i=0}^{d-1} (\log(\hat{f}_i))^4 + \frac{a\lambda}{2}\left(\mathrm{tr}(\log(\hat{\mathbf{F}}))\right)^4. \tag{4}$$

The coefficient a is approximately 6.254 and is obtained by minimizing the L^2-norm of the difference between the quartic function with the original logarithm function over the interval $[0.25, 1]$.

The above model gives visually pleasing results for the explicit time integration scheme. Semi-implicit scheme, where plasticity is treated as a post-process after the elastic response is resolved implicitly, has been shown to suffer numerical cohesion problems [Klár et al. 2016]. We propose a *unilateral* version of the quartic energy density function which mitigates this problem and improves the visual result of [Tampubolon et al. 2017], namely

$$\begin{aligned}\hat{\psi}(\hat{\mathbf{F}}) = a\mu \sum_{i=0}^{d-1} (\log(\hat{f}_i))^4 H_{\{\log(\hat{f}_i)<0\}}(\log(\hat{f}_i)) + \\ \frac{a\lambda}{2}\left(\mathrm{tr}(\log(\hat{\mathbf{F}}))\right)^4 H_{\{\mathrm{tr}(\log(\hat{\mathbf{F}}))<0\}}(\mathrm{tr}(\log(\hat{\mathbf{F}}))),\end{aligned} \tag{5}$$

where H symbolizes the indicator function.

We use the Drucker-Prager plasticity yield function with a rigorously derived return mapping algorithm corresponding to the quartic model. Unlike the model in [Tampubolon et al. 2017], our unilateral elasticity does not require additional parameter tuning and much more closely approximates the visual behavior of the original model from [Klár et al. 2016]. The details of our algorithm are explained in supplemental technical document [Gao et al. 2018b].

7 MORE RESULTS

In addition to the benchmarks, we also demonstrate the efficiency of our GPU implementation and the efficacy of our new heat discretization and the semi-definite sand model with several simulation demos. We list the performance and the parameters used in these simulations in Table 1. Note that all particles are sampled using Poisson Disk [Bridson 2007] for uniform coverage. For the Krylov solver, which are typically used for implicit MPM, each iteration consists of a call to the P2G kernel and a call to the G2P kernel. Detailed timings for the solver are shown in Table 2. Others section includes vector addition, inner-product, and other solver operations, the time of which depend on the number of activated voxels. Beside P2G, computing gradient and SVD also take a large amount of computation time in the solver, however, for the "Granulation" example in the Table 2, since sand particles are very sparse, more voxels are activated. As a results, P2G and others become more expensive. G2P becomes more expensive too, because the more voxels data need to be fetched for interpolation. Thus, the percentage of computing gradient and SVD is reduced, because it only depends on the particle number. Using either an atomics-only scattering or optimized gathering, like GVDB does, P2G occupies more than 90% of the solver time and is the bottleneck of the entire simulation. In contrast, our proposed P2G-MLS method is 23× faster than atomic-only scattering and 15× faster than GVDB gathering as shown in Fig. 6. The P2G time is reduced down to around 40% for the solver.

In Fig. 2, eighteen elastic dragons are stacked together in a glass to generate interesting dynamics; and in Fig. 13, two arrays of dragon-shaped jellies are dropped to the ground. We can also simulate a Gelatin Jello bouncing off another larger one in Fig. 3.

Lava is poured to a cool elastic dragon in Fig. 1. Our heat solver is capable of accurately capturing the process of heat transport and phase change. As the temperature of certain parts of the dragon increases, the dragon liquefies. Finally, we demonstrate in Fig. 14 that our new granular material model manages to produce visually pleasing dynamics with a semi-implicit solver.

7.1 Memory Footprint

Time performance is not the only concern of our implementation. The memory consumption should also be dealt with appropriately, especially in high-resolution animations. In our simulator, the memory budget is partitioned into three categories.

Particle. This part of the memory is used to store all particles' attributes. Its size grows linearly with the particle count and the number of attributes in each particle.

Grid. This part of the memory is used for the sparse grid structure, where the actual amount of memory in use has a linear correlation with the number of occupied GSPGrid blocks. The worst case degenerates to a uniform grid. In our test cases, we mostly set the total amount of memory allocated to be 60% to that of the uniform grid.

Auxiliary. Logically speaking, this part consists of two blocks of memory. Since we use spatial hashing for particle sorting and block topology construction, we need a hash table to perform the task. Its capacity shares a linear correlation with the number of cells in use (64× that of sparse GSPGrid blocks). To reduce hash collision conflicts as well as saving memory, the coefficient is set to 64 (not

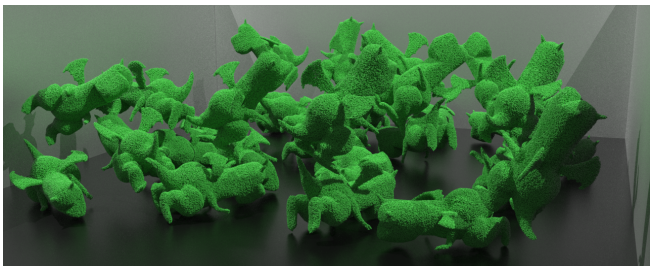

Fig. 13. **How to collide your dragon.** Two arrays of elastic dragons colliding with each other.

the same meaning as before) as a trade-off. Note that the key value of the hash table is a 64-bit integer. The other trunk of memory works as the storage for other intermediate computations, including the ordering of particle positions, velocities, deformation gradient **F**, etc. Its size is linear with the number of particles.

In the actual implementation, the total amount of auxiliary memory set in advance for these intermediate computations is decided by the maximum memory in use at the same time. The hash table is only used for particle ordering and page topology construction which are the two beginning tasks of each time step. In the rest steps, this hash table is no longer needed.

As a result, up to 2.4GB is spent on the cube example with 7M particles, 900K cells, and 17K GSPGrid blocks, of which the majority is particle-related, while the memory budget for the dragon collision example with 95K particles, 6K cells, and 500 blocks is 70MB.

8 LIMITATIONS AND FUTURE WORK

Our work focuses at porting the transfer kernels of the MPM pipeline to the GPU with the least amount of performance compromises. In addition to our data representation via (G)SPGrid, our analysis identified the overhead associated with a scatter-approach as a significant hindrance, due to the fine-grained atomic operations necessary to avoid hazards. Rather than avoiding the scatter paradigm, we introduced an approach that drastically minimizes the need for such atomics, improving parallel efficiency.

A number of the design choices that led to our demonstrated performance gains also carry some associated limitations that we consciously commit to. Our decision to mimic the design of the CPU-oriented SPGrid data structure in our GPU counterpart allows for implementations on the respective platforms to use similar semantics and maximize code reuse. However, the explicit use of the virtual memory system in the CPU version of SPGrid allows for computational kernels to be implemented (at reasonable, albeit not fully optimal efficiency) with computations performed at per-node granularity or, more realistically, SIMD-line granularity; for example, accessing a stencil neighbor of any individual grid node can be done at a reasonable cost, without any set-up overhead. On the GPU, however, such computations can only reach high efficiency if performed at a larger scale, e.g. at block granularity, since the overhead of fetching the neighboring blocks of the one being processed needs to be borne for each kernel invocation. This is not a prohibitive limitation, as performing computations at block granularity is by-and-large a necessity for efficiency for any similar GPU kernel.

GSPGrid apparently lacks a defining feature of CPU/SPGrid, its use of the virtual memory system and translational faculties, i.e. the TLB. One would hope that this can be a momentary shortcoming, and future GPUs and associated APIs will provide more direct access to virtual memory and address translation, on the GPU. But even in the current version, our implementation, even if it feels more like a tiled grid, is trivially convertible from/to a CPU/SPGrid structure. Also, the main benefit that CPU/SPGrid draws from its affinity to the virtual memory system is its ability to deliver competitive performance even if one grid index is processed at a time – on the GPU, we are de facto forced to conduct operations at warp/block granularity, so the design of fetching a neighborhood of blocks prior to kernel application might have been the best performing implementation, even with a virtual memory-assisted data structure.

In addition, for a CPU implementation of SPGrid, accessing a grid node that has not been referenced before is an operation that can be done without any requisite setup or pre-processing (any page-faults that might occur are handled transparently). On the GPU, however, our need to explicitly allocate all active blocks necessitates that the set of all active indices be fully known before their data storage can be allocated and accessed. Once again, we regard this as a reasonable limitation, since the frequency at which the topology of the computational domain changes is small relative to the computational cost of operating on such data during MPM simulation. Also, since the maximum domain size is determined by the number of valid bits of a 64-bit offset and the resolution of each cell, it should be enough for most simulations.

Finally, GPU MPM simulations are still limited by the smaller amount of on-board memory, and it would be an interesting investigation to explore multi-GPU methods, or heterogeneous implementations to circumvent the size limitation.

While our optimization strategies greatly utilize computational resources on the GPU, high fidelity MPM simulations are still far from being real-time. This is largely due to the strict CFL restriction on time step sizes especially in high resolution. It would be interesting future work to further combine additional algorithmic acceleration of MPM time stepping with our GPU framework. We would also explore possibilities with spatially adaptive GSPGrid following Gao et al. [2017] for superior performance.

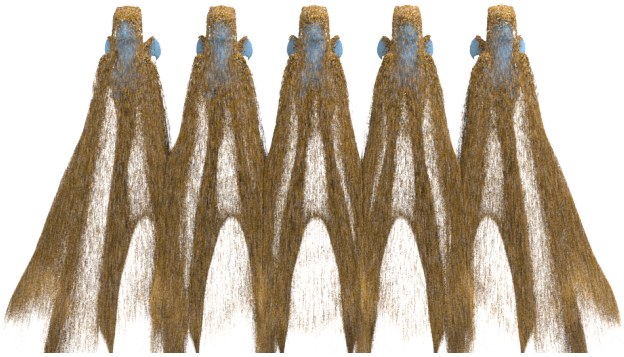

Fig. 14. **How to granulate your dragon.** Granulated dragons fall on elastic ones. This simulation contains 6.7 million particles on a 512^3 grid at an average 39.4 seconds per $48Hz$ frame.

ACKNOWLEDGMENTS

We thank Professor Min Tang for offering all the resources needed and giving his full support for Xinlei Wang in this paper. We also thank Joshuah Wolper for narrating the accompanying video. This work was supported in part by NSF Grants CMMI-1538593, IIS-1253598, IIS-1755544, IIS-1763638, CCF-1533885, CCF-1813624, CCF-1812944, a gift from Awowd Inc., a gift from SideFX, and NVIDIA GPU Grants. Kui Wu is supported in part by University of Utah Graduate Research Fellowship.

REFERENCES

M. Aanjaneya, M. Gao, H. Liu, C. Batty, and E. Sifakis. 2017. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Trans Graph* 36, 4 (2017), 140.

J. Brackbill. 1988. The ringing instability in Particle-In-Cell calculations of low-speed flow. *J Comp Phys* 75, 2 (1988), 469–492.

R. Bridson. 2007. Fast poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 Sketches*. Article 22.

R. Bridson. 2008. *Fluid simulation for computer graphics*. Taylor & Francis.

W-F. Chiang, M. DeLisi, T. Hummel, T. Prete, K. Tew, M. Hall, P. Wallstedt, and J. Guilkey. 2009. GPU acceleration of the Generalized Interpolation Material Point method. *Symp App Accel High Perf Comp* (2009).

J. Chu, N. B. Zafar, and X. Yang. 2017. A Schur complement preconditioner for scalable parallel fluid simulation. *ACM Trans Graph* 36, 5 (2017), 163:1–163:11.

G. Daviet and F. Bertails-Descoubes. 2016. A semi-implicit material point method for the continuum simulation of granular materials. *ACM Trans Graph* 35, 4 (2016), 102:1–102:13.

Y. Dong and J. Grabe. 2018. Large scale parallelisation of the material point method with multiple GPUs. *Comp and Geo* 101 (2018), 149–158.

Y. Dong, D. Wang, and M. F. Randolph. 2015. A GPU parallel computing strategy for the material point method. *Comp and Geo* 66 (2015), 31–38.

Y. Fang*, Y. Hu*, S. Hu, and C. Jiang. 2018. A Temporally Adaptive Material Point Method with Regional Time Stepping. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '18)*. Eurographics Association. (*Joint First Authors).

M. Gao. 2018. *Sparse Paged Grid and its Applications to Adaptivity and Material Point Method in Physics Based Simulations*. Ph.D. Dissertation. University of Wisconsin, Madison.

M. Gao, A.P. Tampubolon, X. Han, Q Guo, G. Kot, E. Sifakis, and C. Jiang. 2018a. Animating fluid sediment mixture in particle-laden flows. *ACM Trans Graph* 37, 4 (2018).

M. Gao, A. P. Tampubolon, C. Jiang, and E. Sifakis. 2017. An adaptive Generalized Interpolation Material Point method for simulating elastoplastic materials. *ACM Trans Graph* 36, 6 (2017).

M. Gao, X Wang, K Wu, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang. 2018b. Supplemental Document: GPU Optimization of Material Point Methods. (2018).

Theodore Gast, Chuyuan Fu, Chenfanfu Jiang, and Joseph Teran. 2016. *Implicit-shifted Symmetric QR Singular Value Decomposition of 3x3 Matrices*. Technical Report. University of California Los Angeles.

N. K. Govindaraju, I. Kabul, M. C. Lin, and D. Manocha. 2007. Fast continuous collision detection among deformable models using graphics processors. *Comp Graph* 31, 1 (2007), 5–14.

N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. C. Lin, and D. Manocha. 2005. Interactive Collision Detection Between Deformable Models Using Chromatic Decomposition. *ACM Trans Graph* 24, 3 (2005), 991–999.

Q. Guo, X. Han, C. Fu, T. Gast, R. Tamstorf, and J. Teran. 2018. A Material Point Method for thin shells with frictional contact. *ACM Trans Graph* 37, 4 (2018).

T. Harada, S. Koshizuka, and Y. Kawaguchi. 2007. Smoothed particle hydrodynamics on GPUs. In *Comp Graph Int*, Vol. 40. SBC Petropolis, 63–70.

R. K. Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proc of High Perf Graph*. Eurographics Association, 109–117.

C. Horvath and W. Geiger. 2009. Directable, high-resolution simulation of fire on the GPU. *ACM Trans Graph* 28, 3 (2009), 41:1–41:8.

Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans Graph* 37, 4 (2018).

P. Huang, X. Zhang, S. Ma, and H. K. Wang. 2008. Shared memory OpenMP parallelization of explicit MPM and its application to hypervelocity impact. *Comp Mod in Eng and Sci* 38 (2008), 119–148.

C. Jiang, T. Gast, and J. Teran. 2017a. Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM Trans Graph* 36, 4 (2017).

C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin. 2015. The affine particle-in-cell method. *ACM Trans Graph* 34, 4 (2015), 51:1–51:10.

C. Jiang, C. Schroeder, and J. Teran. 2017b. An angular momentum conserving affine-particle-in-cell method. *J Comp Phys* 338 (2017), 137–164.

G. Klár. 2014. Speculative Atomics: A Case-Study of the GPU Optimization of the Material Point Method for Graphics. In *GPU Technology Conference*.

G. Klár. 2017. Blasting Sand with CUDA: MPM Sand Simulation for VFX. In *GPU Technology Conference*.

G. Klár, J. Budsberg, M. Titus, S. Jones, and K. Museth. 2017. Production ready MPM simulations. In *ACM SIGGRAPH 2017 Talks*. Article 42, 42:1–42:2 pages.

G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran. 2016. Drucker-prager elastoplasticity for sand animation. *ACM Trans Graph* 35, 4 (2016), 103:1–103:12.

H. Liu, N. Mitchell, M. Aanjaneya, and E. Sifakis. 2016. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Trans Graph* 35, 6 (2016).

Justin Luitjens. 2014. Faster parallel reductions on Kepler. *Nvidia* (2014).

A. McAdams, A. Selle, R. Tamstorf, J. Teran, and E. Sifakis. 2011. Computing the singular value decomposition of 3× 3 matrices with minimal branching and elementary floating point operations. *University of Wisconsin Madison* (2011).

K. Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans Graph* 32, 3 (2013), 27.

S. G. Parker. 2006. A component-based architecture for parallel multi-physics PDE simulation. *Fut Gen Comp Sys* 22, 1 (2006), 204 – 216.

D. Ram, T. Gast, C. Jiang, C. Schroeder, A. Stomakhin, J. Teran, and P. Kavehpour. 2015. A Material Point Method for viscoelastic fluids, foams and sponges. In *In Proc Symp Comp Anim*. 157–163.

K. P Ruggirello and S. C. Schumacher. 2014. A comparison of parallelization strategies for the material point method. In *11th World Cong on Comp Mech*. 20–25.

R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis. 2014. SPGrid: A Sparse Paged Grid structure applied to adaptive smoke simulation. *ACM Trans Graph* 33, 6, Article 205 (2014), 205:1–205:12 pages.

G. Stantchev, D. Dorland, and N. Gumerov. 2008. Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. *J Par Dis Comp* 68, 10 (2008), 1339 – 1349.

A. Stomakhin, R. Howes, C. Schroeder, and J. Teran. 2012. Energetically consistent invertible elasticity. In *In Proc Symp Comp Anim*. 25–32.

A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle. 2013. A material point method for snow simulation. *ACM Trans Graph* 32, 4 (2013), 102:1–102:10.

A. Stomakhin, C. Schroeder, C. Jiang, L. Chai, J. Teran, and A. Selle. 2014. Augmented MPM for phase-change and varied materials. *ACM Trans Graph* 33, 4 (2014), 138:1–138:11.

D. Sulsky, S. Zhou, and H. Schreyer. 1995. Application of a particle-in-cell method to solid mechanics. *Comp Phys Comm* 87, 1 (1995), 236–252.

A. P. Tampubolon, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth. 2017. Multi-species simulation of porous sand and water mixtures. *ACM Trans Graph* 36, 4 (2017).

M. Tang, Z. Liu, R. Tong, and D. Manocha. 2018. PSCC: Parallel Self-Collision Culling with spatial hashing on GPUs. *Proc ACM on Comp Graph Int Techn* 1, 1 (2018), 18:1–18.

M. Tang, R. Tong, R. Narain, C. Meng, and D. Manocha. 2013. A GPU-based streaming algorithm for high-resolution cloth simulation. *Comp Graph Forum* 32, 7 (2013), 21–30.

M. Tang, H. Wang, L. Tang, R. Tong, and D. Manocha. 2016. CAMA: Contact-Aware Matrix Assembly with unified collision handling for GPU-based cloth simulation. *Comp Graph Forum* 35, 2 (2016), 511–521.

H. Wang. 2018. Rule-free sewing pattern adjustment with precision. *ACM Trans Graph* 37, 4 (2018).

X. Wang, M. Tang, D. Manocha, and R. Tong. 2018. Efficient BVH-based collision detection scheme with ordering and restructuring. In *Comp Graph Forum*, Vol. 37. Wiley Online Library, 227–237.

R. Weller, N. Debowski, and G. Zachmann. 2017. kDet: Parallel constant time collision detection for polygonal objects. In *Comp Graph Forum*, Vol. 36. 131–141.

J. Wretborn, R. Armiento, and K. Museth. 2017. Animation of crack propagation by means of an extended multi-body solver for the material point method. *Comp and Graph* (2017).

K. Wu, N. Truong, Yuksel C., and Hoetzlein R. 2018. Fast fluid simulations with sparse volumes on the GPU. *Comp Graph Forum* (2018).

Y. Yue, B. Smith, C. Batty, C. Zheng, and E. Grinspun. 2015. Continuum foam: a material point method for shear-dependent flows. *ACM Trans Graph* 34, 5 (2015), 160:1–160:20.

Y. Zhang, B. Solenthaler, and R. Pajarola. 2008. Adaptive sampling and rendering of fluids on the GPU. In *Proc of the Fifth Eurographics / IEEE VGTC Conf on Point-Based Graph*. 137–146.

Y. Zhang, X. Zhang, and Y. Liu. 2010. An alternated grid updating parallel algorithm for material point method using OpenMP. *Comp Modeling in Eng and Sci* 69, 2 (2010), 143–165.

Y. Zhu and R. Bridson. 2005. Animating sand as a fluid. *ACM Trans Graph* 24, 3 (2005), 965–972.