A Massively Parallel and Scalable Multi-GPU Material Point Method

XINLEI WANG∗, Zhejiang University and University of Pennsylvania
YUXING QIU∗, University of California, Los Angeles and University of Pennsylvania
STUART R. SLATTERY, Oak Ridge National Laboratory
YU FANG, University of Pennsylvania
MINCHEN LI, University of Pennsylvania
SONG-CHUN ZHU, University of California, Los Angeles
YIXIN ZHU, University of California, Los Angeles
MIN TANG, Zhejiang University
DINESH MANOCHA, University of Maryland
CHENFANFU JIANG, University of Pennsylvania

Fig. 1. Crushing concrete. Our system enables this concrete crushing simulation (inspired by the hydraulic press) on a single workstation with 4 NVIDIA Quadro P6000 GPUs. This simulation contains 93.8 million particles on a 1024^3 grid, achieving a 3.9 min/frame performance. (Left) A concrete-style render. (Middle) Coloring by GPU. (Right) Coloring by the plastic volumetric strain for visualizing the damage propagation.

Harnessing the power of modern multi-GPU architectures, we present a massively parallel simulation system based on the Material Point Method (MPM) for simulating physical behaviors of materials undergoing complex topological changes, self-collision, and large deformations. Our system makes three critical contributions. First, we introduce a new particle data structure that promotes coalesced memory access patterns on the GPU and eliminates the need for complex atomic operations on the memory hierarchy when writing particle data to the grid. Second, we propose a kernel fusion approach using a new Grid-to-Particles-to-Grid (G2P2G) scheme, which efficiently reduces GPU kernel launches, improves latency, and significantly reduces the amount of global memory needed to store particle data. Finally, we introduce optimized algorithmic designs that allow for efficient sparse grids in a shared memory context, enabling us to best utilize modern multi-GPU computational platforms for hybrid Lagrangian-Eulerian computational patterns. We demonstrate the effectiveness of our method with extensive benchmarks, evaluations, and dynamic simulations with elastoplasticity, granular media, and fluid dynamics. In comparisons against an open-source and heavily optimized CPU-based MPM codebase [Fang et al. 2019] on an elastic sphere colliding scene with particle counts ranging from 5 to 40 million, our GPU MPM achieves over 100x per-time-step speedup on a workstation with an Intel 8086K CPU and a single Quadro P6000 GPU, exposing exciting possibilities for future MPM simulations in computer graphics and computational science. Moreover, compared to the state-of-the-art GPU MPM method [Hu
et al. 2019a], we not only achieve 2x acceleration on a single GPU but our kernel fusion strategy and Array-of-Structs-of-Array (AoSoA) data structure design also generalizes to multi-GPU systems. Our multi-GPU MPM exhibits near-perfect weak and strong scaling with 4 GPUs, enabling performant and large-scale simulations on a 1024^3 grid with close to 100 million particles with less than 4 minutes per frame on a single 4-GPU workstation and 134 million particles with less than 1 minute per frame on an 8-GPU workstation.

CCS Concepts: • Computing methodologies → Parallel algorithms.

Additional Key Words and Phrases: Numerical methods, parallel computing, GPU

ACM Reference Format:

1 INTRODUCTION

The Material Point Method (MPM) provides significant potential and opportunities to exploit parallelism on modern computing architectures. To date, most work on MPM performance has focused on how to thread the algorithm on conventional CPUs and, to a lesser extent, has attempted to exploit domain decomposition via the Message Passing Interface (MPI). This line of work includes threading particle and grid operations as well as handling the transfer of data between particles and grids, which often results in a bottleneck when parallelized. With the advent of modern accelerator architectures such as GPUs, enough memory and bandwidth are available on the accelerator to perform MPM simulations with the significant number of particles and grid cells needed for generating expansive and high-resolution visual scenes. These new accelerator performance capabilities, including advances in native support for scalable atomic operations on floating-point numbers, results in the ability to perform a relatively large number of computations in a relatively small amount of computing time on a single GPU.

However, the memory and compute power of a single GPU is not limitless. To the best of our knowledge, no prior work has attempted to develop a performant algorithm for MPM that utilizes multi-GPUs in a shared memory context. Given numerous multi-GPU platforms being deployed by vendors both in server and workstation configurations, algorithm development for multi-GPUs will enable us to perform even larger-scale simulations at a significantly reduced computing time on what could be considered commodity hardware. Re-designing MPM algorithms for multi-GPUs is non-trivial. First, as a hybrid simulation method, MPM involves complex operations on particles, grids, and the transfer of data between them. Compared to developing a scalable single GPU algorithm, algorithms utilizing multi-GPUs require inter-GPU communications to program the majority of these operations. Second, MPM simulations usually target scenarios with explosions, fractures, highly deformable solids, and fluids. For such highly dynamic problems, the particle population will fluctuate in time as a function of space and therefore incur load imbalance when multiple devices are used.

To further increase the computational power available to perform MPM simulations in both single- and multi-GPU execution contexts, we make three novel contributions. First, we reformulate the conventional GPU-based MPM pipeline with a fused G2P2G kernel function, which not only enables both single- and multi-GPU performance gains, but is also generalizable to prior MPM designs [Fang et al. 2019; Wolper et al. 2019]. Secondly, we develop a specialized Array-of-Structs-of-Array (AoSoA) particle data structure tailored for our G2P2G kernel utilizing the delayed-ordering technique that maximizes bandwidth efficiency. Finally, we propose a domain-decomposition-invariant computation scheme tailored for multi-GPUs, which significantly reduces the additional memory overhead due to PCIe connections among GPUs. As a result, our method outperforms the heavily optimized state-of-the-art single-GPU MPM implementation [Gao et al. 2018b; Hu et al. 2019a] with a 2x speedup and achieves almost linear scaling on multi-GPUs. Moreover, we accomplished large-scale MPM simulations with truly enormous particle and grid cell counts.

We organize this paper as follows. We review related work in Section 2, serving as the basis for our comparisons to the state-of-the-art. In Section 3, we introduce our improved single-GPU algorithm and outline the kernel fusion procedure and data structure details. In Section 4, we present the new multi-GPU algorithm and include a discussion of memory management and communication, while details on the implementation of our code are provided in Section 5. In Section 6, we present results on an extensive selection of benchmarks using a variety of materials. We also include an analysis of both strong and weak scaling of our algorithm as a function of the number of GPUs, which shows significant performance improvements over the state-of-the-art in GPU implementations as well as significant performance gains when using the GPU algorithm relative to a highly optimized CPU implementation. Finally, in Section 7, we conclude the paper with a discussion of the limitations of our new algorithm and the resulting avenues for future work.

2 RELATED WORK

2.1 HPC-based Simulations in Computer Graphics

Parallelized Solvers. The rapid development of modern CPU and GPU architectures makes it possible to accelerate physics-based simulation by parallelizing existing algorithms using threads, domain decomposition, or some combination thereof. A basic approach to parallelism executes an algorithm using multiple threads on multiple CPU cores on a single node, supported by shared memory programming models such as Intel TBB [Willhalm and Popovici 2008] and OpenMP [Dagum and Menon 1998]. Recent examples include Li et al. [2019], which performs domain decomposition within an optimization time integrator for CPU-based parallel evaluation and factorization of subdomain Hessians.

To achieve even better performance, researchers have developed parallel simulation algorithms for the GPU, which enables more floating-point operations on a per-Watt and per-dollar basis when compared to traditional multi-core CPU architectures. In literature, parallelization of large-scale simulations in fluid dynamics, such as Eulerian fluids [Chentanez and Müller 2011, 2013; Cohen et al. 2010; Pfaff et al. 2010], Lagrangian fluids [Amada et al. 2004; Goswami et al. 2010; Macklin et al. 2014; Vantzos et al. 2018; Winchenbach et al. 2016], and the hybrid Eulerian-Lagrangian solvers [Chentanez et al. 2015; Wu et al. 2018], have all been implemented on a single GPU. For GPU simulations of solid mechanics, Gao et al. [2018b] and
Hu et al. [2019a] implement the high-performance Moving Least Squares MPM [Hu et al. 2018], and Bernstein et al. [2016] and Hu et al. [2019a] explore Finite Element Method (FEM) parallel methods. Due to the ever-increasing demand for computational resources and new hardware releases by vendors, developing multi-GPU solutions is an inevitable trend for physics-based simulations to utilize modern computing hardware effectively. Recent work, such as multi-GPU-based Smoothed Particle Hydrodynamics (SPH) [Dominguez et al. 2013; Rustico et al. 2012; Verma et al. 2017; Xiong et al. 2013], FEM [Li et al. 2020], and parallelized Poisson equation solvers [Ament et al. 2010; Liu et al. 2016], has demonstrated the plausibility of physics-based simulation on multi-GPU platforms.

Another stream of high-performance physics-based simulation utilizes distributed platforms, i.e., cloud-based simulation. Early work commonly makes use of MPI to assign computing tasks to distributed nodes automatically. To better adapt to large topological changes that can occur during a simulation, methods for fluid load balancing in cloud-based simulations are proposed [Mashayekhi et al. 2018; Shah et al. 2018], showing significant potential to achieve high-performance distributed fluid animations.

Efficient Data Structures. From the Eulerian viewpoint, the MPM simulation domain is represented by a discretized structured grid where the volumetric data involved is often spatially sparse in large-scale 3D simulations due to dynamic particle populations. This fact has inspired extensive studies on hierarchical and sparse data structures [Hoetzlein 2016; Liu et al. 2018; Museth 2013; Setaluri et al. 2014] to create efficient data access patterns that mitigate the effects of sparsity. For instance, OpenVDB [Museth 2013], one of the most popular sparse storage schemes in computer graphics, dynamically arranges blocks of a grid in a hierarchical manner similar to B+ tree. Hoetzlein [2016] extends this idea further on GPU and proposes GVDB Voxels with an efficient memory pooling architecture to support dynamic topology changes. Alternatively, SPGrid [Gao et al. 2018b; Setaluri et al. 2014] has proven to be a promising data structure in both MPM [Aanjaneya et al. 2017; Hu et al. 2018] and other fluid simulations [Aanjaneya et al. 2017; Liu et al. 2016; Setaluri et al. 2014]. Additionally, methods such as spatial-temporal coherent spatial hashing are also explored to take advantage of the spatial sparsity [Tang et al. 2016; Wang 2018; Weller et al. 2017]. Recently, Hu et al. [2019a] introduces the Taichi programming model, which exposes high-level interfaces for developing and processing spatially sparse multi-level data structures and benefits researchers by eliminating redundant work in data and performance management.

On the other hand, from the Lagrangian viewpoint, particle information is generally unstructured and stored in an Array-of-Structure (AoS) [Hu et al. 2019a] or Structure-of-Array (SoA) [Gao et al. 2018b] compact layout. SoA promotes coalesced memory accesses of particle data when sequential threads access sequential memory addresses. However, the particles need to be re-sorted after each time step to maintain such an efficient data access pattern [Gao et al. 2018b]. AoS is less efficient in gather/scatter operations such as serialization, where long strides in memory are needed to access all data for a single particle, resulting in the use of multiple memory pages. In contrast, AoS maps more readily to the concept of a particle and performs well in cases of un-coalesced memory access patterns due to the locality of the data for a single particle. However, such a memory layout prevents coalesced reads and writes of particle data, thereby significantly inhibiting both GPU and vectorized CPU performance when coalescing is possible. To exploit both the advantages mentioned above and mitigate the disadvantages, we propose an MPM-centric Array-of-Structs-of-Array (AoSoA) data structure for better performance, which possesses the qualities of both AoS and AoS. Inspired by the Hierarchical Particle Buckets introduced by Hu et al. [2019a] and Bailey et al. [2013], we store particles’ data in a hierarchical manner with AoSoA. The particles are reorganized in low-level bins and high-level block-buckets to conserve the efficiency of both the memory access and the data transfer.

2.2 The Material Point Method in Computer Graphics

Introduction. By Sulsky et al. [1994, 1995], MPM is an extension of Hybrid-Fluid-Implicit-Particle (FLIP) [Brackbill and Ruppel 1986; Zhu and Bridson 2005] from fluid animation in hydrodynamics to general elastoviscoplastic materials simulation in solid mechanics. As one of the most promising discretization choices in physics-based simulation, MPM has been used for simulating numerous materials and diverse phenomena. Prior work includes snow [Gaume et al. 2018; Stomakhin et al. 2013], granular materials [Daviet and Bertails-Descoubes 2016; Gao et al. 2018b; Klár et al. 2016; Zhao et al. 2019], viscoelastic solids [Fang et al. 2019], cloth [Fei et al. 2018; Guo et al. 2018; Jiang et al. 2017; Montazeri et al. 2019], hair [Fei et al. 2018; Guo et al. 2018; Jiang et al. 2017], and non-Newtonian fluids and foam.
[Nagasawa et al. 2019; Ram et al. 2015; Yue et al. 2015, 2018]. Additionally, other complex phenomena have been simulated with MPM including melting [Gao et al. 2018b; Stomakhin et al. 2014], baking [Ding et al. 2019], topological changes and fracture [Wang et al. 2019; Wolper et al. 2019; Wretborn et al. 2017], multiple-material interaction [Gao et al. 2018a; Han et al. 2019; Hu et al. 2018; Tambubolon et al. 2017; Yan et al. 2018], frictional contact and collision [Ding and Craig 2019], etc. Recently, GPU-based acceleration [Gao et al. 2018b; Hu et al. 2019a], as well as spatially [Gao et al. 2017; Yue et al. 2018] and temporally [Fang et al. 2018] adaptive methods have been proposed to improve the computational efficiency of MPM.

Prior work on GPU MPM has focused on the design of GPU-tailored data structures for both particles and grids, as well as the corresponding mathematical operations to achieve better performance; each sub-step is redesigned for GPU (largely using CUDA up to this point). For instance, both Gao et al. [2018b] and Hu et al. [2019a] reduce write conflicts during the Particles-to-Grid (P2G) transfer, either by CUDA warp-level reductions [Gao et al. 2018b] or the random-shuffling of particles inside each block [Hu et al. 2019a]. As reported in these papers, using GPUs can considerably improve performance compared to traditional CPU-based MPM.

2.3 Data Structures and Simulations in HPC

AoS. Particle data structures are largely responsible for CPU and GPU performance as they dictate memory access patterns when parallelizing codes via threading or vectorization. The most commonly adopted memory layouts in HPC are AoS and SoA. Specifically, in terms of particle data layouts, AoS stores all particle data components (e.g., mass, each velocity direction, etc.) in separate arrays, ensuring coalesced memory access when reading/writing the same component of adjacent particles. However, when performing non-coalesced operations like particle-grid transfers, additional sorting methods are required to maintain particle order to guarantee that consecutive thread indices access consecutive particle indices [Gao et al. 2018b]. In contrast, AoS reduces the need for sorting in non-coalesced operations, since its improved memory locality has better performance when randomly accessed. However, the same data components of adjacent particles are no longer adjacent in memory [Hu et al. 2019a], resulting in a non-coalesced data access pattern even when coalescing would otherwise be possible. To take advantage of both the AoS and SoA layouts, researchers have proposed AoSoA to achieve both coalescing/vectorizing data access patterns whenever possible and to improve performance via memory locality when it is not [Wald 2010; Weber and Goesle 2014]. Section 3.2 discusses the implementation of AoSoA in greater detail.

HPC Simulation Frameworks. For scientific simulations in HPC, accelerators are already being adopted broadly with a number of the current top supercomputers leveraging GPU hardware to achieve the majority of their performance [TOP500.org 2019]. In these types of supercomputing configurations, thousands of accelerators are combined with a high-speed interconnect with the goal of reaching exascale-class levels of floating-point operations in the next few years. To achieve portability across the variety of accelerator architectures in use in modern supercomputers, several programming models, libraries, and frameworks have been developed to allow for the manipulation of data structures (e.g., AoS vs. SoA) and parallel loop patterns based on the underlying hardware. Examples of performance portability programming models include Kokkos [E. et al. 2014] and its derivative libraries Cabana [Slattery et al. 2019], a portable library for writing multi-GPU/particle simulations via the AoSoA data structure as well as multi-GPU grid-based simulations which can be used to implement hybrid particle-in-cell algorithms such as MPM. Other examples include SMILEI [Derouillat et al. 2018], an open-source multi-purpose Particle-In-Cell (PIC) implementation that has been applied to a wide range of physics studies, from astrophysical plasma to relativistic laser-plasma interaction.

An analysis of the accelerated machines on the TOP500 list, as mentioned above, and a review of the computational patterns in libraries (such as Kokkos and Cabana) reveal that multi-GPU programming on such machines is relegated mainly to a single GPU per MPI rank. Such a programming model allows for a more straightforward description of parallelism and more accessible programming. However, in the case of many simulation algorithms such as MPM, it forces the application more quickly into the strong scaling limit by further subdividing the problem into smaller pieces. By developing a multi-GPU shared memory programming model in this work, we aim to gain additional performance on modern supercomputers by reducing the number of subdomains needed for parallelization, thus increasing the number of GPUs per MPI rank and reducing the dependence on the performance of the network, including its bandwidth and latency. The multi-GPU advancements in this work are particularly important for machines such as Summit [Facility 2018] as a subset of the GPUs on each compute node has a significantly faster local interconnect than the PCI connection and therefore would strongly benefit from the MPI-free algorithm presented here.

3 IMPROVED SINGLE-GPU ALGORITHM

Before introducing our algorithmic improvements, we first summarize the essential steps of a conventional first-order MPI time integration scheme for incremental dynamics from $t^n$ to $t^{n+1}$ ($\Delta t = t^{n+1}-t^n$).

1. **Particles-to-Grid (P2G).** Transfer mass and momentum from particles to grid nodes: \( \{m_p, m_p \mathbf{v}_p\} \rightarrow \{m_i, m_i \mathbf{v}_i\} \);
2. **Grid Update.** Update grid velocities with either explicit or implicit time integration: \( \mathbf{v}_g^{n+1} \rightarrow \mathbf{v}_g^{n+1} \);
3. **Grid-to-Particles (G2P) and Particle Advection.** Transfer velocities from grid nodes to particles, evolve particle strains, and project particle deformation gradients for plasticity (if any). Update the particle positions with their new velocities: \( \{\mathbf{v}_i^{n+1}\} \rightarrow \{\mathbf{v}_i^{n+1}, f_{\mathbf{p},i}^{n+1}\}, \{p_p, p_p^{n+1}\} \rightarrow \{p_p^{n+1}\} \);
4. **Partition Update.** Maintain the sparse data structure topology by updating the active-block array and the mapping from block coordinates to array indices.

Typically, each particle has several attributes including mass \( m_p \), position \( x_p \), velocity \( \mathbf{v}_p \), deformation gradient \( F_p \), initial volume \( v_0^p \), and the affine matrix \( C_p \), which is the same as the velocity derivative matrix in MLS-MPM [Hu et al. 2018]. On the grid, each node generally stores the grid mass \( m_i \) and the momentum \( m_i \mathbf{v}_i \), from which the nodal velocity \( \mathbf{v}_i \) can be calculated.
Although both GPU- and CPU-based SPGrid use inside each block [Hu et al. 2019a]. Although each kernel is highly reductions [Gao et al. 2018b] or by randomly shuffling particles P2G adopted GPU-tailored data structures for particles and grids and re-
update steps (G2P). When parallelizing MPM algorithms, the general concern about the performance is the transfer operations between particles and grids, i.e., P2G and G2P. These sub-steps become even more crucial to the performance of implicit schemes where significantly more transfer operations are required. Below, we present two techniques to accelerate the transfer operations: 1) Grid-to-Particles-to-Grid (G2P2G), an innovative and fused algorithmic kernel, and 2) Array-of-Structs-of-Array (AoSoA), a new application of a particle data structure with an associated parallel loop strategy.

3.1 G2P2G

Similar to many PIC/FLIP-based solvers, the MPM method uses particles to represent discrete Lagrangian elements of the simulated continuum material and employs the Eulerian background grid as the auxiliary scratchpad to compute spatial derivatives and apply boundary conditions. Within a conventional MPM formulation, the particle states are the primarily evolved quantities. When parallelizing the MPM algorithm, the computations in all the sub-steps (i.e., P2G, grid update, G2P, particle advection, and partition update) are implemented in separate GPU kernels. Prior methods adopt GPU-tailored data structures for particles and grids and reduce write-conflicts during P2G, either through CUDA warp-level reductions [Gao et al. 2018b] or by randomly shuffling particles inside each block [Hu et al. 2019a]. Although each kernel is highly optimized, the synchronization of the grid state required by the grid update incurs the separation of kernels, hindering the GPU MPM performance. This limit calls for additional treatments.

To further reduce the latency on modern GPU architectures, re-ordering the traditional time-stepping strategy and combining several kernels are necessary. In each traditional MPM time step, particle quantities have to be streamed in and out of the GPU global memory for multiple times, i.e., in P2G and G2P. Unlike the GPU MPM kernels implemented in Gao et al. [2018b] where Fp is updated at the end of the G2P kernel, Hu et al. [2019a] reorders pipeline by moving the update of Fp to the beginning of the G2P kernel before the P2G transfer to reduce the redundant particle data accesses. With this modification, the evolved Fp can be reused immediately inside the P2G kernel, thus removing the operations to write and reload the updated Fp to and from the GPU global memory in both the current G2P kernel and the next P2G kernel. Using a similar strategy, we could further reorder the traditional MPM time step and reformulate a new kernel for better efficiency.

We start by analyzing the data dependencies among adjacent MPM sub-steps. As shown in the left column of Fig. 4, we observe some order constraints on data dependencies and execution orders of the sub-steps: 1) The P2G must be finished before the grid update, and the G2P is performed after all the grid states have been evolved. 2) The partition update, wherein the particle-grid mapping and the sparse grid data structure are maintained, depends only on the results of G2P, i.e., the advected particle positions. 3) The P2G transfer relies on the particle-grid mapping, i.e., particles need to know to which grid nodes they should raterize to, which leads to the dependency between the partition update in the current time step k and the P2G in the next time step k + 1. The first two observations exhibit strict data dependencies, which are unchangeable to ensure correct computations. The third one, however, is a weak dependency, since the particle-grid mapping can be staggered differently. Therefore, we can reformulate the execution order of the sub-steps for better performance should the strict data dependencies were preserved.

Following the above analysis, we devise a novel G2P2G kernel by grouping the G2P in time step k and the P2G in time step k + 1 together; see Fig. 4 for a graphical illustration. Specifically, during the G2P, transferring the velocity \( v_p \) and any other higher-order velocity modes of the particles can be interpolated from grids to update particle positions and deformation gradients. When grouping the G2P and the P2G together, these interpolated attributes can be referenced immediately for both the particle updates and the

Fig. 3. Candy bowl. We show how to gather a bowl of candies by pouring 6786 candies from a tube. This simulation runs on 4 GPUs with 23M particles and grid resolution 1024 \( \times \) 1024 \( \times \) 512. Each frame is simulated with approximately 4 seconds, demonstrating the acceleration achieved by our algorithm to simulate and generate large-scale scenes that cannot be efficiently processed using traditional MPM on smaller computing systems.
the grid blocks accessed by the G2P and the following P2G kernel, we design an off-by-two mapping strategy, making it possible to reorganize the time step without sacrificing the performance during the P2G transfer. Below, we present the technical details needed to adopt this new G2P2G pipeline.

**Particle-Grid Offset.** In general, the “scratchpad” pattern is critical to the performance of transfer operations; it refers to a software-managed local data buffer stored in shared memory in the context of GPU computing. For the P2G kernel, this buffer stores the grid attributes, i.e., mass, and momentum, to which particles will rasterize. For the G2P kernel, on the other hand, it stores the attributes of grid nodes from which the particle states would be interpolated. Instead of using a direct mapping between particles and blocks, traditional GPU MPMs use an off-by-one staggering strategy [Gao et al. 2018b; Hu et al. 2019a]. In detail, a staggered mapping between particles and grid blocks with a one-cell-distance is applied to the P2G and the P2G kernel. In this way, each transfer kernel requires a small shared memory buffer with only \(2 \times 2 \times 2\) grid blocks loaded, as shown in the left-side of Fig. 5. Without such a staggering, \(3 \times 3 \times 3\) grid blocks (\(3 \times 3 \times 2\) in 2D) will be needed, increasing the cost of both memory storage and the data accessing.

However, in the G2P2G, the off-by-one staggered mapping between particles and grid nodes cannot be used as it is impossible to keep the assumption that particles would only touch \(2 \times 2 \times 2\) grid blocks during transfers, since we now advect the particles during the G2P2G kernel execution. We solve this problem with an off-by-two staggered mapping, tailored for our G2P2G pipeline. Overall, the local buffer size remains the same as in prior off-by-one staggered mapping [Gao et al. 2018b; Hu et al. 2019a], i.e., \(2 \times 2 \times 2\) grid blocks, with each block containing \(4 \times 4 \times 4\) grid nodes. In detail, bounded by a Courant–Friedrichs–Lewy (CFL) condition, particles would never move more than one-cell distance during the particle advection. Therefore, the grid cells that particles may write to during the P2G transfer would not extend by more than one cell in each 3D direction. When enforcing the particle-grid mapping with the off-by-two strategy, the touched grid blocks would not change for the P2G after the previous G2P and the particle advection. Therefore, the G2P2G pipeline reformulation does not increase the shared

---

**Fig. 5. Different staggered mappings.** Each square represents a cell in space, marked with a color that indicates the block it is located in. During the transfer, particles represented by circles contribute properties to the background grid. (Left) The conventional GPU MPM pipeline adopts an off-by-one staggered mapping between blocks and particles for more efficient use of the shared memory. (Right) The G2P2G pipeline adopts an off-by-two strategy: particles from a block should be located at least two-cell distance from the border of the arena. Such a design ensures the particles to stay in the same blocks after CFL-bounded advection in the G2P2G.

---

next momentum transfer from particles to grids, converting these quantities to temporary variables within the kernel instead of arrays allocated in GPU global memory; the only particle attributes that need to be preserved are the mass, positions, and deformation gradients. With such a G2P2G reformulation, the new MPM pipeline inverts the traditional MPM time step by regarding the grid states as the primarily evolved quantities in each time step, with particles treated as intermediate integration points instead. At a high level, this G2P2G reformulation not only eliminates two transfer kernel launches and two particle data accesses for each time step, which significantly improves the performance but also reduces the particle storage. Note that, in addition to refactoring an explicit time step as presented in this work, the G2P2G approach could also be applied to implicit MPM schemes where the transfer process can take up to 90% of the wall time of a given simulation.

As for the particle-grid mapping strategy, traditional GPU MPM solvers [Gao et al. 2018b; Hu et al. 2019a] employs an off-by-one particle-grid mapping, wherein each particle block only touches \(2 \times 2 \times 2\) grid blocks in both the P2G and the G2P transfer kernels. After the particle advection, the particles may move out of their original particle blocks, and the next P2G could then write to a different set of \(2 \times 2 \times 2\) grid blocks. Although the partition update kernel may remap the particles to grids to ensure the P2G still loads only \(2 \times 2 \times 2\) grid blocks in the next time step, the partition update and the P2G only possess a weak dependency; i.e., the correctness of the calculation would still be guaranteed if the next P2G is executed immediately after the G2P without updating the partition. What does change is that the data accessed in the P2G kernel may need to involve more grid blocks. To eliminate the influence of particle advection on
memory usage or the data-accessing cost. Note that if CFL condition is violated, the premise for the \(G2P2G\) pipeline reformulation will no longer be valid. Thus, the execution of the \(G2P2G\) kernel could fail due to the out-of-bound shared memory access. If such a situation happens, one needs to re-run the solver with a shorter stepping time until the CFL condition is satisfied.

Compute \(dt\). The time step size \(dt\) for MPM evolution should be carefully chosen under the restriction of CFL condition to preserve the numerical stability while at the same time as large as possible to accelerate the simulation process. In order to satisfy both requirements, the maximum velocity of particles is typically used to compute \(dt\). However, since the state of other particles cannot be inferred during the execution of a single \(G2P2G\) kernel thread, retrieving such a global quantity inside the \(G2P2G\) kernel is impossible. As a substitute, we use the maximum velocity of the grid nodes, which can be computed before entering the \(G2P2G\) kernel. Since the particle velocities are interpolated from the surrounding grid nodes, the maximum velocity of particles will not be larger than the maximum grid velocity, and therefore the CFL restriction will be conserved. Moreover, this method is more computationally efficient in \(dt\) estimation since the number of grid nodes is much less than the number of material particles. Although this approach estimates a more conservative \(dt\), experimental results show little difference in the computed \(dt\) (less than 1%) between the computation performed with the maximum velocities of grid nodes and particles.

3.2 AoSoA

Particle data layouts and the corresponding memory access patterns also significantly influence performance, since the particle attributes constitute the majority of the simulation data. In general, for a gather-style transfer, the particle memory throughput is at least one order of magnitude larger than the throughput of the grid data, making it impossible to cache all the particle data in the limited GPU shared memory. However, it is feasible to cache the grid attributes in the corresponding \(G2P\) kernel. For a scatter-style transfer, on the other hand, each particle is commonly assigned to one specific thread, making particles invisible to each other. It is, therefore, more meaningful to cache the grid data instead of the particle attributes to the shared memory. In both cases, inside the \(G2P\) or the \(P2G\) kernel, there is at least a one-time reading from or writing to the GPU global memory to access the particle data, which cannot be cached for better performance. Therefore, optimizing the efficiency of particle data accesses from GPU global memory becomes one of the most significant factors when maximizing performance.

Although both state-of-the-art approaches [Gao et al. 2018b; Hu et al. 2019a] use the GPU-tailored SPGrid variant for grid storage, they adopt fundamentally different particle data structures and algorithmic strategies. Gao et al. [2018b] stores particle attributes in an SoA layout and devises a delayed-reordering technique to maintain the particle order; without reordering, the change of the spatial distribution of particles may lead to an insufficient GPU cache line utilization and cause performance degradation. To get rid of the cost of the particle reordering, Hu et al. [2019a] uses an AoS layout, making the performance less sensitive to the particle order. Nevertheless, the performance is still limited by the non-coalesced read/write of particle attributes from/to the GPU global memory.

To exploit the advantages of both SoA and AoS layouts without compromising performance, we devise an AoSoA data structure to store particle attributes. The particles are grouped according to their positions, such that particles mapping to the same block can be gathered together in the memory. We adopt an AoS structure to store the particle attributes inside each group, while the particle groups are organized using an AoS structure. With such a design, the proposed AoSoA particle data structure has the following advantages:

- As long as the \(SoA\) group size is a multiple of the CUDA warp size, each warp of threads can access (read and write) particle data in a coalesced manner to ensure bandwidth efficiency.
- The particles are grouped according to their positions, and the particle groups are organized in an \(AoS\) layout. Therefore, each block (a \(4 \times 4 \times 4\) cell size in our pipeline) of particles resides in contiguous memory, easier for faster migration among multi-GPUs. Note that the \(SoA\) layout does not possess such property as particle attributes are stridden across the GPU memory. Such a design suits better for the proposed \(G2P2G\) pipeline, wherein each CUDA block handles only one particle block.
- By organizing particles inside each particle block with a finer granularity, we can reduce memory usage by making each particle block to occupy a minimal amount of memory to accommodate the particles inside; see details in the binning strategy paragraph.

**Particle Bins.** To devise an appropriate particle data structure that possesses these properties, we introduce the concept of particle bins, inspired by the designs of SPGrid [Setaluri et al. 2014] and Hierarchical Particle Buckets [Bailey et al. 2013; Hu et al. 2019a].
One intuitive idea is to group particle data in particle blocks such that particles that belong to the same grid block are gathered together. In a single particle block, the particle then becomes the basic unit, with the particle attributes corresponding to the grid channels in the conventional SPGrid. However, compared to the grid block, the particle block would suffer from the large granularity and the uncertainty of the in-use number of particles. In particular, the number of particles residing in a single block is generally orders of magnitude larger than the number of grid nodes, and each particle usually contains more attributes than a grid node. Thus, the actual size of a particle block could be much larger than a grid block. Additionally, the number of particles inside a particle block changes dynamically throughout the simulation, causing memory waste and additional bookkeeping operations.

To remedy these problems, we further group the particles inside a single block into particle bins; the size of a particle bin can be customized as needed. For performance considerations, we recommend setting the bin size to be a multiple of the thread group size on a given GPU architecture. For example, one can set the bin size as 32, which is the size of a CUDA warp on an NVIDIA GPU.

As illustrated in Fig. 7, particle data is organized in an SoA layout within each particle bin. In this way, coalesced global memory accesses are ensured with the CUDA 32-, 64-, or 128-byte transactions that are aligned to these sizes. Another advantage of using particle bins instead of a monolithic SoA particle block is related to the page management in the virtual memory system. For example, a particle bin containing 64 particles, with each particle owning 16 float-type attributes, consumes a 4KB memory space. In contrast, the particle block with the same setting would consume a space much larger than the 4KB configuration. Although the actual page size in CUDA might differ from the CPU page setting in practice, the particle binning strategy still provides the potential to better utilize the automatic CUDA unified virtual memory management.

The mapping from a block to its particles is implemented through the Hierarchical Particle Bucket design. Specifically, particle attributes and particle indices are stored separately in particle blocks and particle buckets, both in a $4 \times 4 \times 4$ block granularity. Each particle is reached hierarchically through the block index and the local index inside the block. In practice, an upper bound of the particle bucket size is predetermined statically at compile-time, the maximum number of bins inside a block is predetermined by the bucket size when compiling, and the number of bins that each block contains can also be decided at run-time before execution. However, as illustrated in Fig. 8, such a uniformly allocated particle-block memory may cause a significant memory waste. To further reduce memory usage, we count the number of bins in-use and establish a mapping from the block ID to the bin ID through a lightweight exclusive scan.

**Binning Strategy.** There are typically two strategies to reduce the write conflicts in the $G_{2P}$ kernel:

- Group particles by cells, reduce at warp-level, perform a single shared memory atomic increment per warp, and perform a single global memory atomic increment per block [Gao et al. 2018b].
- Leave particles unsorted to reduce the chances of atomic-write conflicts and avoid the warp-level reduction [Hu et al. 2019a].

The first method imposes restrictions on the order of the particles, which cannot be satisfied in the context of the $G_{2P2G}$ pipeline; particles may advect to the neighbor cells after the $G_{2P}$ transfer in the $G_{2P2G}$ kernel, breaking the cell-based sorted order.

Adopting ideas from the second strategy, we use a pseudo-coloring procedure and collect particles from different cells within a particle block to build the particle bins. The algorithm is outlined in Alg. 1, which stops when there are not enough particles left to form a bin. With this strategy, particles inside a single bin are forced to write to different nodes unless the bin is formed after satisfying the stopping condition; i.e., there exist at least two particles from the same cell inside this bin. As a result, the chance of write conflicts occurring within a warp is significantly reduced. Note that the warp aggregated atomic increment is required to ensure the correctness of the Alg. 1; see Adinets [2014] for more implementation details.

**Update Particles.** Inside the $G_{2P2G}$ kernel, the maintenance of the particle structure must be performed after the particle advection. To ensure the execution correctness of the proposed $G_{2P2G}$ pipeline, we adopt a double buffer strategy for both particles and grids; i.e., the $G_{2P2G}$ kernel reads from and writes to different particle/grid buffers. To maintain the particle structure, a naive scheme can be adopted to update the particle attributes in place while the particle orders are rearranged in an extra kernel incurring additional overhead. Following the delayed-ordering [Gao et al. 2018b], we postpone the particle reordering in the $G_{2P2G}$ kernel to the next time step.

Specifically, the updated particle attributes are written back to the particle blocks in the coalesced manner, and the particle indices are inserted into the particle buckets according to their updated positions. In the following time step, we determine the particle attributes in the previous particle block buffer from the indices saved in the current particle bucket. Theoretically, the particle block ID...
**Algorithm 1:** Distribute particles from cell buckets to block bucket

```plaintext
// ppc: particles per cell
// ppb: particles per block

**Given:** cell_buckets, ppcs, maxppc, blockid, cellid

**Output:** block_bucket, ppbs

**Procedure:**

```plaintext
laneid ← cellid mod warpsize
ppc ← ppcs[cellid]

pidic ← 0 // pidic denotes local particle index in cell

while pidic < maxppc do
    if pidic < ppc then
        // pidib denotes local particle index in block
        pidib ← aggregate_atomicadd[laneid, ppbs[blockid]]
        block_bucket[pidib] ← cell_bucket[pidic]
    end if
    syncthreads_in_block()
    pidic ← pidic + 1
end while
```

and its local index inside the block would change after the particle advection. However, we do not update the hierarchical particle indices immediately after updating particle positions. Instead, we compute the new indices from the advection vector and their original location; as indicated by the CFL bound, the particles will move at most at a one-cell-distance in each time step. In practice, we use −1, 0, or +1 to indicate the particle’s movement in x, y, or z direction to form the 3D advection vector (i.e., one specific vector from a set of 27 possibilities). Given the previous block ID and the advection information, the new particle indices are then uniquely determined by a spatial hash with a 32-bit integer.

4 MULTI-GPU PIPELINE

Using multi-GPUs for MPM simulations affords significantly larger simulations and shortens the overall simulation time. To extend from using a single GPU to running on multi-GPUs, we divide the whole simulation domain into partitions according to the device number and assign one partition to one GPU device. The load balancing is one of the essential considerations when distributing partitions for multi-GPU applications. Depending on the dynamics of the simulation, the same partitioning scheme could result in drastically different performances on various problems. Ultimately, the parallel efficiency of multi-GPUs is primarily determined by 1) how large the halo region compared to the whole partition, and 2) how equally the partitions are distributed on all devices. Here, we focus on arranging the computations once the partitioning strategy is confirmed.

Additionally, we maintain the sparse spatial information according to particle positions at each time step. The partition on each device is maintained through a list of activated blocks that cover all particles. Since the particles may rasterize to grid blocks, which can be halo blocks and shared by multi-GPUs, the attributes on grid blocks must be synchronized after the P2G transfer. Therefore, in addition to partitioning strategies, efficient utilization of multi-GPUs for MPM also needs to consider:

- **Halo Block Tagging:** tag the blocks that overlap partitions on other devices (i.e., the halo blocks).

### 4.1 Multi-GPU Static Partitioning by Particles (MGSP)

MGSP is an ideal option for solid simulations, including elastic jellies, sand, and other granular materials, due to the stable halo distribution of solids. Since the overall shape of solid models remains intact even under large deformations, the halo regions typically reside on the model surfaces. Even when significant fractures happen (see examples in Figs. 1 and 6), the halo regions still only occupy a small portion of the whole partition.

Carrying out both halo block tagging and halo block merging relies heavily on multi-GPU communication. The latency of the related operations relies highly on the underneath hardware setup. In most consumer-level machines, multi-GPU devices are connected via the slow PCI-Express x16 Gen 3, which may lead to high communication latency. Fortunately, nearly all CUDA devices with compute capability of 1.1 or higher can concurrently perform the memory copies and computing kernels. Therefore, it is possible to hide the latency by overlapping data transfers with computations (i.e., G2P2G) through CUDA streams, as shown in Fig. 9.

**Halo Block Tagging.** For each device, to acquire its intersections with other devices, the coordinates of the active blocks from all the other devices are gathered and then checked in a local hash table. We perform the halo block tagging as an additional step of the MPM algorithm; see Fig. 10. Due to the data dependencies, this step should not be overlapped with other computations. The data size of the active block coordinates increases with the growth of the simulation scale when more blocks are involved. However, in general, such data size is still small, making this additional overhead introduced by multi-GPU extensions insignificant.

**Halo Grid Reduction.** The heavy workload of the G2P2G kernel provides the potential of overlapping the memory copies with the computations; see an illustration in Fig. 9. Based on the halo block tagging results, we split the particle blocks on each device into two groups. One group produces data for halo grid blocks during the G2P2G execution, whereas the other only works with the interior grid blocks. The G2P2G kernel is first launched for grids and particles inside the halo regions. After that, the following two operations are performed simultaneously with different CUDA streams, i.e., 1) the...
Halo grid attributes on each partition are gathered and sent to other partitions, and 2) the G2P2G kernel is evaluated on the particles and grids outside the halo regions on each device. In this way, the overhead of the memory copies among GPUs is masked with the execution for interior particles and grids.

4.2 Multi-GPU Static Partitioning by Space (MGSS)

In an MPM simulation, it is possible that the size of halo regions among multiple partitions grows beyond a threshold, such that the latency of the non-halo G2P2G kernel is not high enough to mask the device-to-device memory copies. This situation is especially common for fluid simulations where fluids can theatrically mix (see Fig. 15 as an example), making halo sizes increasing dramatically as time goes by. In such cases, re-partitioning particles is necessary for load balancing, and statically partitioning by space is a simple yet efficient strategy.

Halo Block Tagging. Unlike in MGSP, the blocks in the halo region in MGSS can be tagged without the knowledge of any other partition. While updating the partition, blocks located in the spatially predefined halo region are directly tagged as halo blocks, and halo regions can be shared by two or more devices depending on the splitting scheme. The handling of the tagged halo grid blocks in MGSS is the same as in MGSP, but the particles moving to partitions on other devices are also migrated in addition to the grid data.

Fig. 10. Halo block tagging. We tag grid blocks with three labels: assigned, transferred, and advected. Assigned grid blocks are the blocks where particles are residing in. Transferred grid blocks contain the grid-nodes that particles may write to, e.g., with quadratic B-spline kernel, each particle may write to neighbor nodes within a three-cell distance. Advected grid blocks represent the blocks where particles may advect to after the G2P and the particle advection. The transferred and the advected grid blocks may contain no particles in the current time step. In the context of multi-GPU MPM, the partition from one device can overlap partitions from other devices. These overlapping regions (i.e., halo regions) may include all three types of grid blocks. The grid data in the halo regions should be shared and synchronized by the corresponding GPUs to ensure the execution correctness.

Halo Particle Migration. Although the overhead of halo tagging in MGSS is avoided, and halo grid reduction in MGSS is similar to the one in MGSP, there is an additional task in MGSS; namely, particles moving out of the current domain must be migrated to the corresponding device. This operation is easily supported by our AoSoA particle data structure since particles are already grouped by blocks, and it is efficient to retrieve these particles before streaming. Furthermore, gathering particles in halo regions in bulk and streaming to other devices are always better than sending the same amount of data in pieces at a time, e.g., particle by particle. Therefore, the same AoSoA particle data structure also specifies the particle buffer array for sending halo data to and receiving data from other devices.

The migration of halo particles in MGSS is inherently more memory intensive than sharing halo grid blocks in MGSP. In general, particles have more quantities compared to grid nodes, and the number of particles inside each particle block is an order of magnitude larger than the number of grid nodes inside each grid block. Consequently, within the same halo region, particle blocks use significantly more memory than grid blocks. Moreover, the number of particle bins at each location near the boundary of a domain is only known after G2P2G kernels in all neighboring partitions are done, which breaks the premise of “compact storage” (Section 3.2). Fortunately, the maximum number of such halo blocks is bounded and known at compile time and is small compared to the whole domain. A simple workaround regarding the number of particle bins is to preserve a space conservatively that is fit for the maximum number of particles specified in the "Hierarchical Particle Bucket.”

5 IMPLEMENTATION

In this section, we provide essential implementation details. More information is included in the supplemental material.

Multi-GPU Communication. Although there are multiple CUDA libraries (e.g., OpenSHMEM [Chapman et al. 2010], NCCL [Nvidia 2019]) for inter-GPU communications, we directly use the low-level memory APIs for better control over the double-buffering scheme and halo communication. The halo data can be manually copied through host, peer-to-peer, GPUDirect, or to exploit the use of Unified Virtual Memory (UVM) and let CUDA handle on-demand requests of halo data in UVM. However, page faults during kernel executions are expensive, and pre-fetching block-by-block before
kernel launching requires many CUDA API calls. Hence, we choose to manually initiate the data transfers through multiple streams.

Memory Consumption. Table 1 summarizes the memory footprint in our pipeline and Hu et al. [2019a]: each particle block contains 4096 particles in total. As shown in Fig. 7, there are 16 bytes for a fluid particle and 52 bytes for a solid particle in our G2P2G pipeline. In contrast, in the conventional pipeline, there are 68 bytes and 100 bytes correspondingly. In our pipeline, the per-particle storage size is reduced substantially, especially in the fluid case, due to particle velocity being a kernel-local quantity. However, we need to maintain two copies of the data structures due to the double-buffer strategy.

Table 1. Memory budget. We compare the number of particle blocks and grid blocks with the block size $4 \times 4 \times 4$. Here, numbers followed by $n$ stand for the number of particle blocks and $m$ for the number of grid blocks. We use 64 as the maximum number of particle-per-cell inside each grid cell; this setting is more than sufficient for most MPM simulations (8 is the typical setting). We have not observed any violations in any examples. Note that the data in the second row represents the total memory cost from the double buffering required by the G2P2G kernel.

<table>
<thead>
<tr>
<th></th>
<th>particle buffer</th>
<th>solid</th>
<th>grid</th>
<th>bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>conventional</td>
<td>278528n</td>
<td>409600n</td>
<td>1024m</td>
<td>16384n</td>
</tr>
<tr>
<td>G2P2G</td>
<td>131072n</td>
<td>425984n</td>
<td>2048m</td>
<td>16384n</td>
</tr>
</tbody>
</table>

Material-dependent Computation. The constitutive model related particle-wise operations (elasticity, plasticity, etc.) are implemented in separate device functions, and the correct function for a specific material to call is automatically handled when utilizing the C++ sum type variant. Thus, different materials are easily supported with little changes in our approach.

Generalizations to Other MPM Methods. Not only is the AoSoA particle data structure compatible with different algorithmic or material choices, but the G2P2G kernel is essentially a general operator evolving the grid state through transfers. With reasonable efforts, the majority of existing MPM methods (including Jiang et al. [2017] and Wolper et al. [2019]) can also be implemented with AoSoA+G2P2G. In the case of implicit MPM, matrix-free linear solvers, as in Gao et al. [2018b], can be implemented directly with the G2P2G fusion strategy used for a single matrix vector product to improve performance. Moreover, our AoSoA+G2P2G design can benefit other hybrid particle-grid simulation methods, such as PIC/FLIP fluids, with better performance, more efficient memory usage, and flexible extension to multi-GPUs.

6 BENCHMARKS AND PERFORMANCE EVALUATIONS

In this section, the fixed corotated constitutive model [Stomakhin et al. 2012] is applied by default for all benchmarks unless stated otherwise. We use microseconds as units for all timings. The codebase used to generate these examples is made publicly available.

For experiments with different materials, we experimented with multiple parameters, which are set as easy-to-set compile-time constants. For instance, the crashing concrete scene is tested with Young’s modulus ranging from 66 to 668 for grid resolution $512 \times 512 \times 512$ and $1024 \times 1024 \times 1024$, since concrete is really a mixture of materials with no absolute stiffness. We choose one of the material parameter settings we tested for the final results and list them in Table 5 for reproduction purposes.

In the following subsections, we start with the single-GPU performance comparison against the state-of-the-art methods. Two ablation studies are presented to analyze the efficacy of the proposed AoSoA+G2P2G design. We then move to multi-GPU settings with discussions of scalability, comparisons of two partitioning strategies, and demonstrations of large-scale simulations.

6.1 Single-GPU Performance

6.1.1 Speedup over State-of-the-art Methods. When comparing with the state-of-the-art method [Hu et al. 2019a], we apply the optimal settings listed in Hu et al. [2019a], i.e., AoS for particles, and SPGrid for grid blocks. Moreover, we set up the following scenes for performance evaluations.

- **dragons.** 775196 particles, $256 \times 256 \times 256$ grid.
- **bomb falling.** 984018 particles, $256 \times 256 \times 256$ grid.

As shown in Table 2, our pipeline reaches around 2x speedup compared to the state-of-the-art approach, Hu et al. [2019a]. Under a more fair setting with the initial sorting of particles in Hu et al. [2019a] disabled, we further achieve a 2.5x speedup. Measured speedups show consistencies on NVIDIA GPUs for both gaming (RTX series) and computing (Quadro series) and for different generations.

Table 2. Single-GPU performance comparison. All candidate single-GPU MPM methods use the MLS-MPM transfer method in explicit time integration. The per-time-step timing results are run on NVIDIA RTX 2080 and Quadro P6000 and gathered after objects hit the ground for better evaluation. The Hu et al. [2019a] benchmark disabled the initial reordering. The dragons scene reduces particles per cell by half.

<table>
<thead>
<tr>
<th></th>
<th>Hu et al. [2019a]</th>
<th>ours</th>
<th>Hu et al. [2019a]</th>
<th>ours</th>
<th>Hu et al. [2019a]*</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>dragons</td>
<td>4.5</td>
<td>3.0</td>
<td>1.3</td>
<td>1.8</td>
<td>2.0</td>
<td>0.9</td>
</tr>
<tr>
<td>dragons*</td>
<td>6.4</td>
<td>6.7</td>
<td>3.4</td>
<td>4.2</td>
<td>6.4</td>
<td>2.5</td>
</tr>
<tr>
<td>bomb falling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also compare the timing against an open-source, heavily optimized CPU-based MPM codebase [Fang et al. 2019] (a SIMD vectorized implementation provided by its authors). The experiment is conducted in an elastic sphere colliding scene with particle counts ranging from 5 to 40 million. On a workstation with an Intel 8086K CPU and a single Quadro P6000 GPU, our GPU MPM achieves 110% to $120 \times$ per-time-step speedup, as summarized in Table 3.

Table 3. Performance comparison between an SIMD implementation vs our GPU pipeline. CPU: Intel 8086K. GPU: Quadro P6000 GPU.

<table>
<thead>
<tr>
<th># of particles</th>
<th>5m</th>
<th>10m</th>
<th>15m</th>
<th>20m</th>
<th>25m</th>
<th>30m</th>
<th>35m</th>
<th>40m</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpus time</td>
<td>5.97</td>
<td>11.91</td>
<td>18.22</td>
<td>27.38</td>
<td>35.38</td>
<td>43.67</td>
<td>51.43</td>
<td>59.23</td>
</tr>
</tbody>
</table>

6.1.2 Ablation Studies.

G2P2G Speedup. We implement Hu et al. [2019a] with the proposed G2P2G kernel. As shown in Table 4, all of the test cases have achieved around 40% speedup, except for the cube case where the model is generated with uniform sampling rather than Poisson sampling. With perfectly balanced particle distribution in the cube case, the negative impact of redundant particle data access pattern in P2G and G2P pipelines is mitigated. Moreover, the G2P2G kernel may lessen the latency-hiding capability [Laine et al. 2013] compared to...
conventional separate transfer kernels (i.e., P2G and G2P), limiting the performance gain in the cube case. In addition to improving performance, the proposed G2P2G pipeline also decreases the storage size required for each particle, making it more favorable for particle migrations in the multi-GPU pipeline.

Table 4. Ablation study. The first timing column is the sum of the timings of P2G and G2P kernels. The timing in the second timing column is measured by replacing P2G and G2P kernels with the proposed G2P2G kernel. The timing in the third timing column is measured by replacing the AoS layout with the proposed AoSoA layout on top of the G2P2G kernel. The speedup is calculated by comparing it with the reference time [Hu et al. 2019a]. Both bomb failing and dragons scenes use irregular geometries; all dragons scenes have the very same geometry but are sampled with different numbers of particles per cell, and bomb failing scene is much denser in space. The cube scene is a uniformly sampled cube with particles ordered. All timings are computed using an NVIDIA RTX 2080 graphics card.

AoS A SoA Speedup. On top of the G2P2G pipeline, we further change the AoS in [Hu et al. 2019a] to our proposed AoSoA layout. As shown in Table 4, the combined improvements enhance the transfer kernel with around 3x speedup without introducing any additional overheads of the maintenance or the storage of particle data.

6.2 Multi-GPU Scalability

The scaling with multi-GPU devices is an essential aspect of evaluating the efficacy and robustness of the algorithm. Ideally, the performance should scale with the number of devices and remain robust when simulating scenes that have different patterns for the halo regions. We perform scaling benchmarks on a workstation with one Intel Core i7-8086K CPU, four NVIDIA Quadro P6000 GPUs, and 64GB RAM assembled on a Z390 motherboard.

Weak Scaling. We assign each GPU device with one giant cube containing 4,096,000 particles. All cubes are either arranged compactly or side-by-side. In the compact layout, each partition shares a certain amount of halo regions with partitions from all the other GPU devices. In the side-by-side layout, each partition is only in contact with at most two neighboring partitions. The weak scaling comparisons are shown in Figs. 12 and 13.

Strong Scaling. Four cubes of the same size that contains 4,741,632 particles are used to form a long cuboid. The scene is even partitioned and assigned to multi-GPU devices. The strong scaling comparisons are shown in Fig. 14.

Results. Taken together, the scaling results indicate that the G2P2G kernel, as the bottleneck of the algorithm, is scaling almost linearly when each GPU is saturated by enough computations. Additionally, our multi-GPU MPM pipeline scales almost perfectly with the increasing number of GPUs. The improved efficiency with respect to memory access and data communication (e.g., fewer attributes stored, coalesced data accessing, and particle data locality) is also preserved in multi-GPU systems.
Table 5. **Parameters and timings.** We summarize the parameters of particle numbers, grid resolutions, $\Delta x$, the average time per frame, and the maximum $\Delta t$ for various experiments described in Section 6.4. These examples are simulated with different materials; material-related information is recorded in the last two columns. Specifically, FC denotes the fixed corotated material, NACC for Non-Associated Cam-Clay, and DP for the Drucker-Prager elastoplasticity. In addition to the basic settings of the material (density $\rho$, Youngs Modulus $E$, and Poisson ratio $\nu$), we also include other material-specific parameters. The material parameters are listed in the following order: 1) FC: $(\rho, E, \nu)$, 2) NACC: $(\rho, E, v, \alpha_s, \beta, \xi, M)$, 3) DP: $(\rho, E, v, f, a, c)$, and 4) Fluid: $(\rho, k, \gamma)$. We recommend review the corresponding papers for further information about parameters.

<table>
<thead>
<tr>
<th>example</th>
<th>particle #</th>
<th>GPU #</th>
<th>grid resolution</th>
<th>ave sec/frame</th>
<th>$\Delta t_{min}$</th>
<th>$\Delta x$</th>
<th>max $\Delta t_{top}$</th>
<th>material</th>
<th>material parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Fig. 2) bomb falling</td>
<td>134,007,186</td>
<td>8</td>
<td>$512 \times 2048 \times 512$</td>
<td>59.56</td>
<td>1/48</td>
<td>$2.71 \times 10^{-5}$</td>
<td>FC</td>
<td>$(100, 3 \times 10^3, 0.2)$</td>
<td></td>
</tr>
<tr>
<td>(Fig. 3) candy bowl</td>
<td>22,900,536</td>
<td>4</td>
<td>$1024 \times 1024 \times 512$</td>
<td>4.15</td>
<td>1/48</td>
<td>$2.10 \times 10^{-4}$</td>
<td>FC</td>
<td>$(100, 3 \times 10^3, 0.2)$</td>
<td></td>
</tr>
<tr>
<td>(Fig. 1) crushing concrete</td>
<td>53,790,217</td>
<td>4</td>
<td>$1024 \times 1024 \times 1024$</td>
<td>236.89</td>
<td>1/240</td>
<td>$1.74 \times 10^{-6}$</td>
<td>NACC</td>
<td>$(2240, 6 \times 10^3, 0.2, -0.01, 0.5, 0.6, 1.85)$</td>
<td></td>
</tr>
<tr>
<td>(Fig. 6) soil falling</td>
<td>52,904,854</td>
<td>4</td>
<td>$512 \times 512 \times 512$</td>
<td>57.38</td>
<td>1/48</td>
<td>$1.65 \times 10^{-5}$</td>
<td>NACC</td>
<td>$(2, 3 \times 10^4, 0.3, -0.006, 0.3, 0.5, 1.85)$</td>
<td></td>
</tr>
<tr>
<td>(Fig. 11) sand armadillo</td>
<td>55,088,474</td>
<td>4</td>
<td>$512 \times 512 \times 1024$</td>
<td>34.39</td>
<td>1/48</td>
<td>$3.58 \times 10^{-5}$</td>
<td>DP</td>
<td>$(20, 1 \times 10^4, 0.4, 30, 0)$</td>
<td></td>
</tr>
<tr>
<td>(Fig. 15) single dam-break</td>
<td>48,608,497</td>
<td>4</td>
<td>$512 \times 2048 \times 512$</td>
<td>15.17</td>
<td>1/240</td>
<td>$1 \times 10^{-5}$</td>
<td>Fluid</td>
<td>$(1000, 4 \times 10^4, 7, 15)$</td>
<td></td>
</tr>
</tbody>
</table>

### 6.3 Partitioning Comparisons

Although MGSP is a perfectly balanced partitioning method in terms of the number of particles, the overhead due to halo block tagging would increase with more GPU devices employed. Moreover, when the size of the halo regions becomes large enough, the memory latency will increase and become the dominant factor compared to the latency of the G2P2G kernel. Such a performance degradation may frequently happen in fluid simulations where fluid may significantly mix together as time goes by, resulting in an increasing number of halo region storages and computations. In such cases, it would be more efficient to use MGSS where the halo region size stays the same throughout the simulation time; we demonstrate the partitioning using MGSS throughout a dam-break scene in Fig. 15.

### 6.4 Large-scale Simulations

We showcase a suite of simulations with various materials to demonstrate the scalability of our multi-GPU MPM algorithm. The following constitutive models with plasticity are implemented to demonstrate the applicability of our methods to diverse materials: 1) fixed corotated [Stomakhin et al. 2012] to simulate elastic jello, 2) Non-Associated Cam-Clay (NACC) [Wolper et al. 2019] to reproduce soil and concrete, 3) Drucker-Prager elastoplasticity [Klar et al. 2016] for sand animation, and 4) weakly compressible fluid [Tampubolon et al. 2017] to generate water. All timings and spatial resolution settings are summarized in Table 5. Additionally, we also provide material related parameter settings for reproduction purposes.

We first demonstrate the scalability of the proposed multi-GPU MPM in Fig. 2, wherein 13,346 bombs fall onto the ground. This example is run on 8 GPUs, with the grid resolution $512 \times 2048 \times 512$, 134M particles, and each frame finished within 1 minute on average. To the best of our knowledge, no prior work has achieved such a large-scale simulation with MPM on with a single machine. In addition to the 8-GPU test, we also evaluate this scene on 4 GPUs with $6,688$ bombs (67M particles). In a 4-GPU context, proper scaling is achieved with each frame simulated in 49.14 seconds on average.

Using the fixed corotated elastic material, we fill the bowl in Fig. 3 with 6,786 candies (23M particles) with each frame finished within 5 seconds on average. In other words, one only needs 20 minutes to obtain the results of a 200-frame simulation with 20M particles, which usually would take several days if only CPU-based MPM algorithms were adopted.

We crush concrete in Fig. 1 with NACC models, showing hydraulic press experiments on a concrete cylinder. The simulation domain is discretized into a $1024 \times 1024 \times 1024$ grid with $\Delta x = 1/1024$, while the concrete cylinder is represented by 93.8M particles. On a 4-GPU workstation, each frame is finished within 4 minutes. Note that only 4 (instead of 8) GPUs are employed to simulate 96M particles, indicating a strong potential of the proposed AoSoA+G2P2G in simulating large-scale scenes with limited memory resources.

Moreover, we further test the same scene with different settings of resolutions, particle numbers, and material parameters. Timing statistics show that it takes only 17 seconds to simulate the same scene with 12M particles and grid resolution $512 \times 512 \times 512$.

As another NACC example, three soil chunks fall, fracture, and mix together in Fig. 6; each frame with 52M particles is finished under 1 minute. In comparison, as reported in Wolper et al. [2019], a NACC example with only 1.67M particles consumes at most 10 minutes on a CPU-based MPM implementation. Similar to Fig. 1, we visualized the NACC-$\alpha$ to indicate the crack propagation.

Sand material is used to create two armadillos smashing together with fine details captured in Fig. 11. This scene has 55.5M particles with grid resolution $512 \times 512 \times 1024$. Simulating each frame takes less than 30 seconds using the proposed multi-GPU MPM pipeline.

We demonstrate a large-scale fluid simulation with the MGSS strategy in a single dam-break experiment, shown in Fig. 15. The topology of the fluid changes substantially as the simulation evolves, resulting in different portions of the fluid to mix together as time goes by. The size of the halo region would increase substantially as the simulation proceeds should we utilize the MGSP strategy; it would lead to significant performance degradation as most of the run-time would be spent in inter-GPU communication. In contrast, with the MGSS strategy, even though different portions of fluid are permeating into each other, the multi-GPU partitions are still relatively well balanced with a fixed-size halo region.

### 7 LIMITATION AND FUTURE WORK

**Limitation.** Our G2P2G kernel inherently requires a double buffer strategy for simultaneous read and write of particle and grid data. This fact could offset some of the savings of memory from the per-particle storage size. Although we use compact storage for particle attributes, their indices are still managed in the corresponding buckets that are pre-allocated with a uniform and conservative size. This design imposes restrictions on more irregular MPM simulations where the number of particles per cell is significantly larger.

**Future Work.** For simplicity, we adopt the “pre-allocation for all” strategy for all spatial data structures specified in our codebase due to the lack of a dedicated allocator. A more customized allocator could provide more flexibility in terms of memory management,
e.g., on-demand allocation. There is also room for improvement in terms of robustness. We will work on an adaptive and unified framework that supports multi-material simulations, including both solids and fluids, and more flexible load balancing by allowing for dynamic re-partitioning of the whole domain, which would change the method of halo-region identification and memory preservation for halo particles. Deploying to distributed systems, e.g., cloud or multi-GPU clusters, is another challenging yet promising direction worth of research efforts.

In the robotics community, we recently observe a growing amount of work that exploits physics-based simulation to facilitate robot learning in navigation [Xie et al. 2019], embodiment mapping [Liu et al. 2019], soft robot locomotion [Hu et al. 2019b], tool-using [Zhu et al. 2019], inferring human utility [Zhu et al. 2016], and causality [Edmonds et al. 2020]. These tasks are traditionally considered to be extremely challenging. With the capability to run large-scale simulations on multi-GPUs with a relatively short simulation time, we expect the robot learning community would start to adopt high fidelity simulations to enable robots acquiring knowledge and skills swiftly with minimal human intervention or supervision.

ACKNOWLEDGMENTS

We thank Yuanming Hu at MIT for useful discussions and proofreading. Feng Gao at UCLA for his help on configuring workstations, and the anonymous reviewers for their valuable comments. X. W. and M. T. were supported in part by the National Key R&D Program of China (2017YFB1102703), and NSFC (61972341, 61972342, 61732015, 61572423). Penn authors were supported in part by ONR MURI N00014-16-1-2007, DARPA XAI N66001-17-2-4029, and M. T. were supported in part by the National Key R&D Program of China (2017YFB1002703), and NSFC (61972341, 61972342, 61572423). Penn authors were supported in part by ONR MURI N00014-16-1-2007, DARPA XAI N66001-17-2-4029, and M. T. were supported in part by the National Key R&D Program of China (2017YFB1002703), and NSFC (61972341, 61972342, 61572423).

REFERENCES


