Received: 20 June 2022 | Revised: 28 September 2022 | Accepted: 4 October 2022

DOI: 10.1002/nme.7139

RESEARCH ARTICLE

WILEY

Leveraging ray tracing cores for particle-based simulations on GPUs

Shiwei Zhao[1] | **Zhengshou Lai**[1,2] | **Jidong Zhao**[1,3]

[1]Department of Civil and Environmental Engineering, The Hong Kong University of Science and Technology, Hong Kong SAR, China

[2]School of Intelligent Systems Engineering, Sun Yat-sen University, Shenzhen, China

[3]HKUST Shenzhen-Hong Kong Collaborative Innovation Research Institute, Shenzhen, China

Correspondence
Zhengshou Lai, Department of Civil and Environmental Engineering, The Hong Kong University of Science and Technology, Hong Kong SAR, China.
Email: laizhengsh@ust.hk

Funding information
Guangdong Basic and Applied Basic Research Foundation, Grant/Award Number: 2022A1515010848; National Natural Science Foundation of China, Grant/Award Numbers: 51909095, 51909289, 11972030; Project of Hetao Shenzhen-Hong Kong Science and Technology Innovation Cooperation Zone, Grant/Award Number: HZQB-KCZYB-2020083; Research Grants Council of Hong Kong, Grant/Award Number: 16211221

Abstract

Neighbor searching is an essential and computationally heavy step in particle-based numerical methods such as discrete element method (DEM), molecular dynamics, peridynamics, and smooth particle hydrodynamics. This article presents a novel approach to accelerate particle-based simulations by leveraging ray tracing (RT) cores in addition to CUDA cores on RTX GPUs. The neighbor search problem is first numerically converted into a general ray tracing problem so that it can be possible to utilize the hardware acceleration of RT cores. A new, general-purpose RT-based neighbor search algorithm is then proposed and benchmarked with a prevailing cell-based one. As a showcase, both algorithms are implemented into a GPU-based DEM code for simulating large-scale granular problems including packing, column collapse and debris flow. The overall simulation performance is examined with varying problem sizes and GPU specs. It demonstrates that the RT-based simulations are 10%–60% faster than the cell-based ones, depending on the simulated problems and GPU specs. This study offers a new recipe for next-generation high-performance computing of large-scale engineering problems using particle-based numerical methods.

KEYWORDS

GPU, hardware-accelerated, neighbor search, particle-based simulation, ray tracing cores

1 | INTRODUCTION

Particle-based numerical methods, such as discrete element method (DEM),[1] molecular dynamics (MD),[2] peridynamics (PD),[3] and smooth particle hydrodynamics (SPH),[4] by the virtue of their mesh-free and highly parallelizable features, are gaining increasing popularity in research and engineering. For the particle-based numerical methods, the motion of a particle is determined by the interactions with its surrounding particles (neighbors). Searching for the neighbors of every particle is a critical step, which can be so computationally intensive that dominates the whole running time of a simulation. Notably, the configuration of particles may change significantly and rapidly, especially for dynamic problems, making the neighbor searching problem computationally heavy in general. For example, it can take up to 80% running time for contact detection and resolution (where the computation of neighbor search is the dominant part for spherical particles) in DEM as reported in the literature.[5,6] Therefore, it is worth the effort to develop new acceleration schemes for these particle-based methods, making them more practical in engineering-scale simulations.

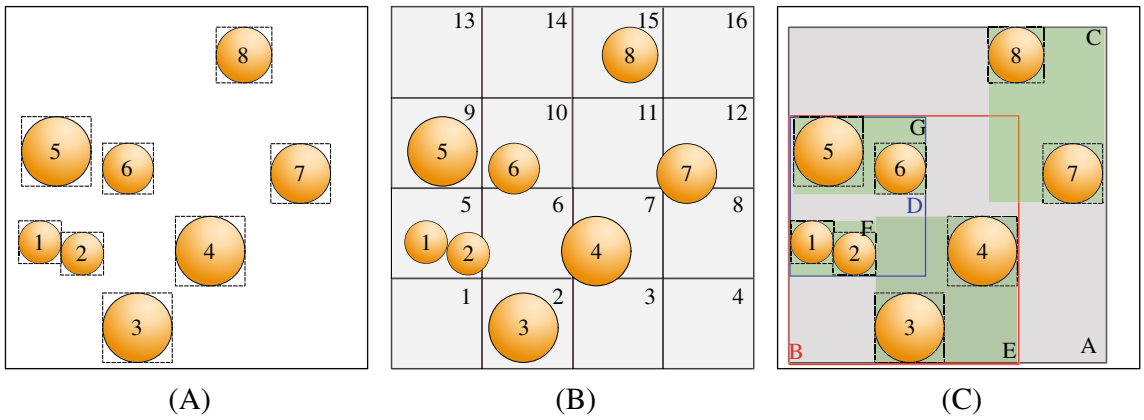

FIGURE 1 Two-dimensional illustrations of: (A) Particles with axis-aligned bounding boxes (AABBs), (B) particles in a uniform grid of cells, and (C) particles organized in a bounding volume hierarchy (BVH).

Indeed, great efforts have been dedicated to algorithms of neighbor search in particle-based numerical methods. Given an N-particle system as exemplified in Figure 1A, neighbor search can be implemented in a brute-force manner, where all the rest $N-1$ particles are queried for each given particle. The brute-force approach is straightforward but has a time complexity of $O(N^2)$, and thus is seldom employed in practical simulations. A prevailing alternative is the so-called sweep and prune algorithm,[7,8] where the lower and upper bounds of axis-aligned bounding boxes (AABBs) are sorted in separate axes, and a particle is regarded as a potential neighbor if its AABB overlaps with that of the given particle in all axes. The sweep-and-prune neighbor search is efficient even for moving particles, and interested readers can find its implementation in some open-source codes, for example, YADE[9] and SudoDEM.[10] Two more prevailing algorithms have also been commonly adopted for neighbor searching: one is cell-based[11,12] as illustrated in Figure 1B and the other is tree-based,[12] especially bounding volume hierarchy (BVH) in Figure 1C. The cell-based neighbor search relies majorly on a uniform grid by which a mapping between particles and grid cells can be obtained efficiently. Hence, searching neighbors of a particle over the entire domain can be reduced by only traversing the particles in the cell of the given particle and its adjacent cells. As for the BVH-based neighbor search, one needs first to build a BVH tree from all AABBs of particles, and then to find the neighbors of a given particle by traversing the corresponding branches and nodes of the BVH tree.

With the rapid development of general-purpose graphics processing unit (GPGPU) computing, a modern GPU can offer tens of thousands of physically concurrent threads for scientific computation beyond the original purpose of image/video rendering. For example, NVIDIA's CUDA (compute unified device architecture) platform provides a scalable programming model for GPU computation with a hierarchical organization of threads. Not surprisingly, such abundant GPU-threads can accelerate particle-based simulations in nature, for example, one thread for one particle. Indeed, there are extensive studies and applications of the parallelization algorithms with CUDA for DEM,[13-16] MD,[17-19] PD,[20-22] and SPH.[23,24] Regarding the algorithms of neighbor search in those particle-based methods, both cell-based and BVH-based algorithms are commonly utilized to leverage the parallel power of GPUs, whilst the sweep and prune algorithm cannot be efficiently implemented on GPUs.

GPUs are naturally more powerful and suitable for processing rendering tasks than general-purpose computing (scientific computing). Prior to NVIDIA's Turing architecture (2019),[25] both rendering and GPGPU computing are empowered by CUDA cores. With the rapid development of computer graphics and meta-universe techniques, NVIDIA has designed additional new specific hardware, namely ray-tracing (RT) cores, for ray-tracing rendering complementary to CUDA cores. Indeed, NVIDIA has released its third-generation RT cores in its recent Ada Lovelace architecture (2022). RT cores are becoming more powerful to fulfill increasing demand in rendering. Although RT cores are originally designed for ray tracing in computer graphics (akin to CUDA cores for the original rendering function of a GPU), their tremendous computational power are driving researches and engineers to explore its potential applications in general-purpose computation beyond ray tracing (akin to CUDA cores for the GPGPU computing). Along this line, Wald from NVIDIA and his collaborators first explored the capability of RT cores for locating points in a tetrahedron mesh.[26] Wang et al.[27] utilized RT cores to trace particles in unstructured meshes for fluid flow in porous media. Zellmann et al.[28] used RT cores to accelerate force-directed graph drawing. Evangelou et al.[29] adopted RT cores for fast radius search in graphic applications that require intensive spatial search operations. The success of RT cores beyond ray tracing implies possibility of applying RT

cores to scientific computing. Regarding particle-based simulations with DEM, MD, PD, and SPH, they have been accelerated much more significantly on GPU than CPU computing systems, but all applications are limited to the utilization of CUDA cores. In these numerical methods, the BVH traversal supported by RT cores may benefit the computationally intensive neighbor search. However, it is not clear to what extent that RT cores can accelerate these particle-based simulations. Hence, it is of interest to propose algorithms for leveraging RT cores in particle-based simulations and estimating the potential performance of using the powerful RT cores together with CUDA cores.

This article presents a novel generalization of the neighbor search problem for DEM, MD, PD, and SPH by using RT cores. The RT-based neighbor search algorithm is proposed along with the cell-based one as a benchmark in Section 2. Both algorithms are then implemented in GPU-accelerated DEM in Section 3, followed by numerical examples including granular packing, column collapse, and dry debris flow in Section 4.

2 | METHODOLOGY

In this section, we generalize the neighbor search problem that are typically involved in particle-based numerical methods and present the proposed RT-based algorithm for neighbor search, along with the conventional cell-based neighbor search algorithm as a comparison.

2.1 | Neighbor search problem

DEM and MD are discrete-based numerical methods, while PD and SPH are continuum-based. In DEM, the overlap between two particles with given sizes (radii) is used to determine inter-particle interaction forces, while cut-off distance is used for MD. As for PD and SPH, an influential range, that is, horizon, is employed to determine interaction weight of a neighbor for a given particle. In these particle-based methods, as shown in Figure 2, a particle j is taken as a potential neighbor of particle i (particle A in the figure) when satisfying the following condition

$$d < (1 + \delta)l, \tag{1}$$

where d is the distance between particles i and j; l is the generalized searching radius, which is equal to the sum of radii of the two particles for DEM, the cut-off distance for MD, and the horizon for PD and SPH; δ is referred to as skin size factor for scaling up the searching radius l and is introduced to reduce the neighbor searching frequency. Neighbor searching is invoked once the accumulative displacement (since last time of neighbor searching) of any particle exceeds its skin size (i.e., δl). Note that for non-spherical particles in DEM, l can be the sum of radii of bounding spheres of the two particles being considered.

2.2 | Cell-based neighbor search algorithm

Cell-based neighbor search approach has been prevalently adopted in particle-based simulations and is briefly introduced herein for the sake of completeness. Interested readers are referred to our previous work[30] where a detailed

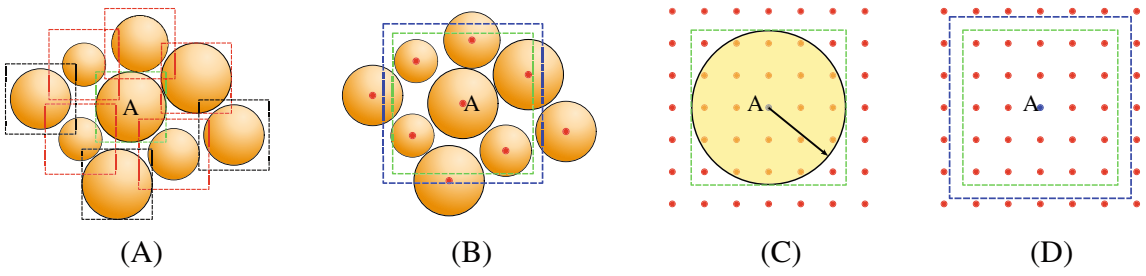

FIGURE 2 (A) Particles with uniform AABBs in DEM or MD, (B) AABB (blue) expanded with respect to the searching region (green) in DEM or MD, (C) horizon-defined searching region (circle) and its AABB (green) of particle A in PD or SPH, and (D) AABB (blue) expanded with respect to the horizon of particle A in PD or SPH.

TABLE 1 Exemplified PC_{N_p} and CP_{N_s}

(Particle id i, PC_i)	(1,5)	(2,5)	(3,2)	(4,7)	(5,9)	(6,10)	(7,12)	(8,15)
(Cell id i, CP_i)	(2,3)	(5,1)	(5,2)	(7,4)	(9,5)	(10,6)	(12,7)	(15,8)

implementation of the algorithm is depicted. Two major steps, namely (i) mapping between particle and cell and (ii) neighbor searching, of the algorithm are presented in the following.

(1) Mapping between particle and cell. Given a set of particle positions X_{N_p} (N_p is particle number), loop over all particles and compute the index (id) PC_i of the cell that particle i belongs to. Similarly, it is necessary to create a cell-particle list CP_{N_s} (N_s is cell number) to link cells to particles. In this work, it is assumed that one particle is mapped to only one cell and the grid cells are larger than the largest neighbor searching distance $(1+\delta)l$ defined in Equation (1). Then, a radix sort with respect to cell ids is performed on the pair of particle-cell ids, so that all particles for a given cell i can be indexed by a look-up list CP_i. Table 1 lists exemplified PC_{N_p} and CP_{N_s} for the configuration in Figure 1B.
(2) Neighbor searching. The neighbors of a given particle i can be searched with the following three steps:
   (i) locating the cell, that is, cell id PC_i, that the particle belongs to;
   (ii) finding the nearest adjacent 26 cells in which the particles are the potential neighbors of the given particle i; and
   (iii) traversing all particles belonging to the cell of interest and its adjacent ones, and performing neighbor check. A neighbor list NL_i is then created and filled with the neighbor particle id j. To facilitate the computation, j is pushed into the sub-lists from left if particle id j is greater than particle id i, and otherwise from right.

In practice, different particles may have different numbers of neighbors and thus require different amount of memory to store the neighbor list. To avoid the overhead due to repeated memory allocation, one can pre-allocate the neighbor list NL_i with a sufficiently large memory segment to cover all possible neighbors. In addition, in most particle-based numerical methods, the particle pair-wise calculations need only to be performed once for each pair of particles. As each pair of particles are neighbors of each other, we only need to check the neighbors of each given particle with particle id less than the id of the given particle; and thanks to the use of two accompanying lists (i.e., sub-list $N_{N_p}^{jgi}$ and sub-list $N_{N_p}^{jli}$), which are introduced to record the counts of neighbors with id j greater or less than particle id i, respectively, the targeted neighbors for a given particle i can be accessed readily.

2.3 | New ray-tracing-based neighbor search algorithm

2.3.1 | Ray-tracing problem

Ray tracing is a lighting technique for modeling light reflection and refraction in the real world, and is prevailing for rendering digital images with extra realism in computer graphics and games. Figure 3A illustrates the main idea of a general ray tracing technique. Each pixel of the rendered image corresponds to a ray starting from the camera. The pixel is then assigned with a specific color, which is computed from the colors of the objects hit by the corresponding ray. During the course of traveling in the scene, a ray may hit several objects sequentially or none, as illustrated in Figure 3B. Typically, there might be tens of thousands of hits (collision detection) for a realistic scene. To facilitate the identifying of a ray path, BVH has been widely adopted. In general, the efficiency of querying a BVH tree is highly dependent on the quality of the tree, which is associated with the implementation complexity. There have been extensive investigations on the BVH-based algorithms on GPUs in past decades.[31,32] Investigation of BVH is beyond the scope of this study, while a ray-tracing neighbor search algorithm will be introduced based on BVH.

2.3.2 | NVIDIA's RT core and OptiX

To accelerate the real-time ray tracing in gaming, NVIDIA has introduced a new type of hardware called RT core in RTX GPUs since the Turing architecture. In an RTX GPU, each streaming multiprocessor (SM) has an RT core, and each RT

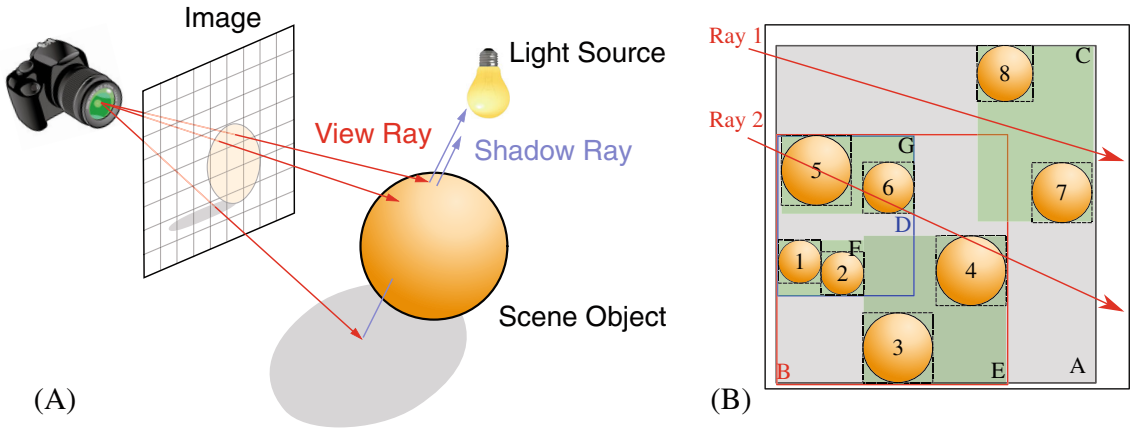

FIGURE 3 Illustration of ray tracing: (A) problem, and (B) accelerating with BVH.

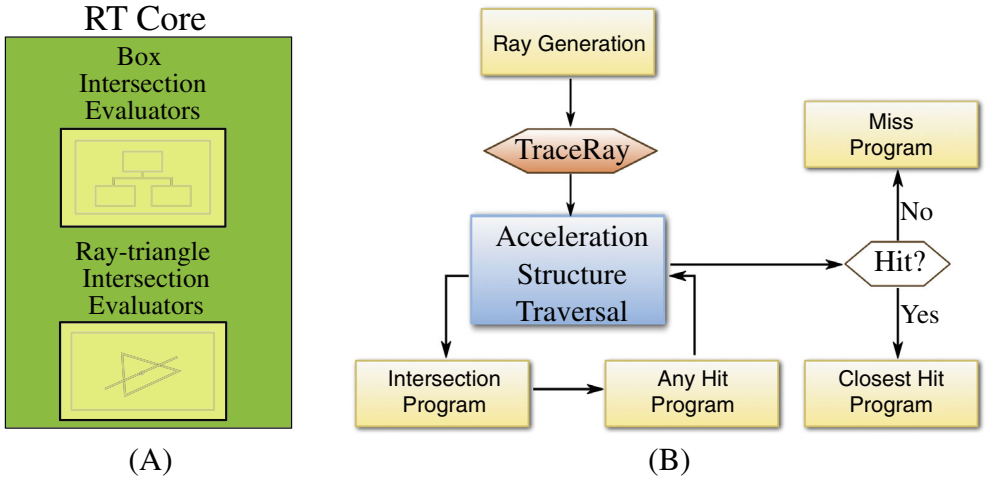

FIGURE 4 (A) NVIDIA's RT core with two physical units, and (B) general ray-tracing workflow in OptiX 7.

core has two physical units: box intersection and ray-triangle intersection evaluators, as illustrated in Figure 4A. As the name suggests, the box intersection evaluator is designed for BVH traversal, while the ray-triangle intersection evaluator is for the solution of a ray hitting a triangle. Accompanied with RT core is newly designed OptiX 7, a ray tracing application programming interface (API) developed by NVIDIA.[33] Figure 4B shows a general ray-tracing workflow in OptiX 7, where yellow boxes denote customized CUDA programs. After launching a ray tracing pipeline, a ray is generated with customized position, length and direction, and then passed to acceleration structure traversal (AST), that is, traversing the BVH of all objects in the scene. Note that AST runs completely on the box intersection evaluator of RT core. A customized intersection program, followed by an any hit program, is invoked once the ray intersects an AABB. A closest hit program is invoked once the ray hits the closest geometry. A miss program is invoked when the ray hits nothing. It is worth noting that OptiX 7 has implemented the art-of-the-state BVH build and traversal algorithms at the driver level, which not only improves the ray tracing performance significantly but also eases the programming of ray tracing.

2.3.3 | Ray-tracing algorithm of neighbor search

The neighbor search problem in particle-based simulations can be converted into a ray tracing problem to leverage the power of RT cores. As shown in Figure 5A, the AABB (green box) of particle A or its horizon region is first expanded as the AABB (blue box) of the neighbor search region. Determining whether a surrounding particle is a potential neighbor of

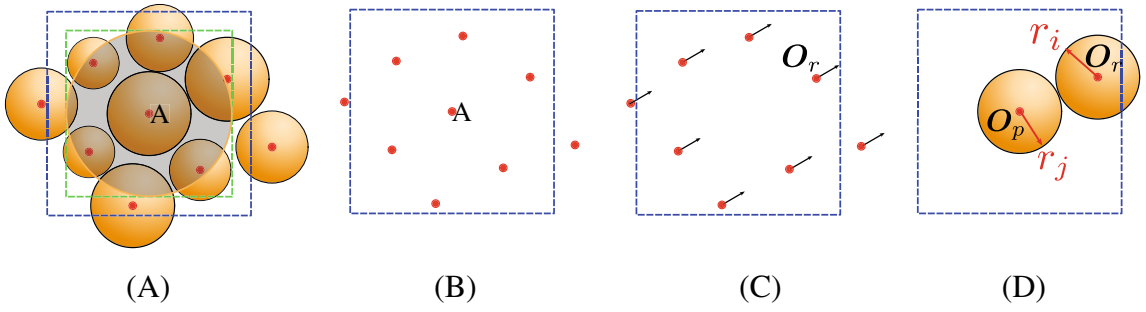

FIGURE 5 (A) Search range of particle A and the corresponding expanded AABB, (B) potential neighbors with centers (red points) and the expanded AABB of particle A, (C) point-AABB intersection as a ray-tracing problem, and (D) further contact resolution for potentially contacting particles.

particle A thus amounts to testing whether the center of the surrounding particle $\boldsymbol{O}_p$ is enclosed by the expanded AABB, referring to Figure 5B. Thereby, the problem can be converted to such a ray tracing one that whether an infinitesimal ray starting from the center (red point) of a given surrounding particle intersects the expanded AABB (blue box) or not, referring to Figure 5C. Given that the ray length is infinitesimal, the ray direction can be arbitrary.

```
optixTrace(gas_handle, center, direction, 0.0f, 1e-16f, 0.0f,
           OptixVisibilityMask( 1 ),
           OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT |
           OPTIX_RAY_FLAG_DISABLE_ANYHIT,
           0,
           0,
           0,
           N_jgi, N_jli ); //N_jgi and N_jli are payloads
```

Listing 1: TraceRay entry of ray generation program in OptiX

Listing 1 shows a snippet of the ray generation program. For each particle, the OptiX API *optixTrace* will launch a ray with a small length of 1×10^{-16}. To identify particles and rays, ray ids are initialized as particle ids. Moreover, both closest hit and any hit are disabled by setting flags *OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT* and *OPTIX_RAY_FLAG_DISABLE_ANYHIT* to improve performance. In OptiX, each ray can have shared data among different programs via setting payloads, while CUDA intrinsics (e.g., shared memory, thread warp) are not allowable. Similar to the cell-based neighbor search approach, two auxiliary lists $N_{N_p}^{jgi}$ and $N_{N_p}^{jli}$ are introduced and passed to *optixTrace* as payloads for particle i (or ray i).

Following the ray generation program, an intersection program will be invoked once there are any ray-box (AABB) intersections for a given ray i during the BVH traversal. Outlined in Algorithm 1 is a pseudocode of such an intersection program for neighbor searching. In general, a surrounding particle is enclosed in the AABB of particle i identified by ray id i as illustrated in Figure 5D once the ray tracing program runs into the ray-box intersection program. However, one more query is conducted at Line 6, referring to Equation (1), to have a better comparison with the cell-based algorithm in Section 2.2. Starting from Line 7, the neighbor list NL_i is constructed in the same manner as the cell-based algorithm except that N_{jgi} and N_{jli} are updated in the payloads of ray i.

Remarks: In OptiX, the customized programs are executed sequentially for each ray that is assigned to a single thread, while a large number of rays can be generated and processed in parallel. To maximize the performance, the neighboring rays should follow similar paths when traversing the BVH. One effective way is to reorder particles following space filling curves such as Morton curve[34] and Hilbert space filling curve.[35] The Morton curve is adopted for ease of implementation but without losing generality in this work. Moreover, the BVH can be constructed via OptiX API but not shown here for brevity. However, special attention needs paying to the quality degradation of the BVH, which would cause a

Algorithm 1. Box intersection program for neighbor searching

Input: ray id i, particle id j, ray position $\boldsymbol{O}_r$, particle position $\boldsymbol{O}_p$, particle radii r_i and r_j.
Output: neighbor lists N_{jgi} and N_{jli}.

```
if i == j then
    return;
N_jgi: ← the first payload;
N_jli: ← the second payload;
d = O_r − O_p;
if ||d|| < (1 + δ)(r_i + r_j) then
    if j > i then
        //push from left;
        NL[i * offset + N_jgi] = j;
        N_jgi: ← N_jgi + 1;
        the first payload: ← N_jgi;
    else
        //push from right;
        NL[(i + 1) * offset − N_jli − 1] = j;
        N_jli: ← N_jli + 1;
        the second payload: ← N_jli;
```

worse BVH traversal, especially for dynamic simulations. Rebuilding the BVH periodically is an effective remedy to avoid this issue.

2.4 | Contact list

It is worth noting that the neighbor list doubly stores the information of neighbors (contact pairs). Hence, the contact pairs are accessible by traversing only half of the neighbor list, for example, neighbors with id j greater than i. Indeed, the list of contact ids can be established by

$$cid = S_i^{jgi} + j, \quad j \in \{0, \dots, N_i^{jgi} - 1\}, \tag{2a}$$

$$S_i^{jgi} = \begin{cases} 0, & \text{if } i = 0, \\ \sum_{k=0}^{i-1} N_k^{jgi}, & \text{otherwise}, \end{cases} \tag{2b}$$

where $S_{N_p}^{jgi}$ is the prefix sum of $N_{N_p}^{jgi}$. Then, a contact list can be built with size N_c given by

$$N_c = S_{N_p-1}^{jgi} + N_{N_p-1}^{jgi}. \tag{3}$$

Note that a backup of the contact list is required once the neighbor list has been updated. With the backup of the contact list, it is straightforward to query the previous contact when the corresponding contact id changes in the updated contact list. Hence, any contact-wise quantities can be solved accordingly, for example, frictional force in DEM that requires an incremental summation, referring to Equation (7).

3 | GPU-ACCELERATED DEM

The proposed RT-based approach for neighbor searching is applied and implemented in a DEM code for validation and performance evaluation. This section presents a brief introduction to DEM and its GPU-based implementation.

3.1 | Brief introduction to DEM

Particle motion and inter-particle contact forces are two essential ingredients of DEM and are introduced briefly below for the completion of presentation. Interested readers are referred to literature[5,10] for more details. In a nutshell, particle motion is governed by the Newton–Euler equation as

$$\boldsymbol{F} = m\dot{\boldsymbol{v}}, \tag{4a}$$

$$\boldsymbol{T} = \boldsymbol{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\boldsymbol{I}\boldsymbol{\omega}), \tag{4b}$$

where $\boldsymbol{F}$ and $\boldsymbol{T}$, respectively, are the force and torque acting on the particle center of mass; $\boldsymbol{v}$ and $\boldsymbol{\omega}$ are the translational and angular velocities, respectively, and the over dot denotes derivation with respect to time; m is the particle mass; $\boldsymbol{I}$ is the moment of inertia around the mass center. Note that the rotation is solved with a reference frame fixed to the particle. For spherical particles, Equation (4b) is reduced to $\boldsymbol{T} = \boldsymbol{I}\dot{\boldsymbol{\omega}}$. The force $\boldsymbol{F}$ and torque $\boldsymbol{T}$ are calculated as

$$\boldsymbol{F} = \boldsymbol{F}^b + \sum_{c \in N_c} \boldsymbol{f}^c, \tag{5a}$$

$$\boldsymbol{T} = \sum_{c \in N_c} \boldsymbol{r}^c \times \boldsymbol{f}^c, \tag{5b}$$

where $\boldsymbol{F}^b$ is the body force; $\boldsymbol{f}^c$ is the contact force at contact c; N_c is the number of particles contacting with the given particle; $\boldsymbol{r}^c$ is the position vector of contact point at contact c with respect to the particle center.

Contact force $\boldsymbol{f}^c$ in Equation (5) is generally decomposed into two orthogonal parts: normal contact force $\boldsymbol{f}_n$ and tangential contact force $\boldsymbol{f}_t$. The force-displacement law in conjunction with the Coulomb friction model is employed as the contact force model at the microscopic scale,[1] given as

$$\boldsymbol{f}_n = -k_n \boldsymbol{d}, \tag{6a}$$

$$\Delta \boldsymbol{f}_t = -k_t \delta \boldsymbol{u}, \tag{6b}$$

and

$$\boldsymbol{f}_t = \begin{cases} \boldsymbol{f}_t' + \Delta \boldsymbol{f}_t, & \text{if } \|\boldsymbol{f}_t' + \Delta \boldsymbol{f}_t\| \le \mu \|\boldsymbol{f}^n\|, \\ \mu \|\boldsymbol{f}^n\| \frac{\delta \boldsymbol{u}}{\|\delta \boldsymbol{u}\|}, & \text{otherwise} \end{cases} \tag{7}$$

where $\boldsymbol{d}$ and $\delta \boldsymbol{u}$ are the penetration vector and the tangential incremental displacement, respectively; $\Delta \boldsymbol{f}_t$ is the incremental tangential contact force; $\boldsymbol{f}_t'$ is the tangential contact force at the previous time step; μ is the coefficient of contact friction. To facilitate energy dissipation in the system, two artificial damping, namely mass damping and viscous damping, are introduced such that

$$\boldsymbol{a} = -\alpha_d \boldsymbol{a} \text{Sign}(\boldsymbol{v}, \boldsymbol{a}), \tag{8a}$$

$$\boldsymbol{f} = 2\bar{\boldsymbol{v}} \beta_d \sqrt{\bar{m} k}, \tag{8b}$$

where α_d and β_d are the coefficients of mass damping and viscous damping, respectively; $\boldsymbol{a}$ and $\boldsymbol{v}$ are the acceleration and the velocity of the particle, respectively; Sign is the signum function with respect to each axis; $\boldsymbol{f}$ and k are the normal/tangential viscous force and the normal/tangential contact stiffness at the contact, respectively; $\bar{m}$ is the mean mass of the two particles, and $\bar{\boldsymbol{v}}$ is their relative velocity at the contact.

3.2 | Framework of GPU-DEM

Algorithm 2 summarizes the pseudocode of generic parallelized DEM on a GPU. For each DEM step/iteration, three main procedures are sequentially executed by launching corresponding CUDA kernels: (a) updating the neighbor list;

(b) computing contact forces for all contacts; and (c) integrating motion of all particles. Note that in the case of spherical particles as considered in this work, updating the neighbor list is more time-consuming than the other two procedures, but it can be executed less frequently via introducing skin size in Equation (1) to cache potential neighbors in the neighbor list. To this end, a switch *updateNL* is flagged to trigger the neighbor list update once any particle moves a distance larger than the skin size since the last update of the neighbor list in the integration procedure of particle motion.

Algorithm 2. Pseudocode of the GPU-DEM workflow

```
Input: Simulation iterations;
foreach iteration do
    if updateNL then
        update reference positions of particles;
        update neighbor list;
        reorder particles if applicable;
    // Computing contact forces for all contacts;
    foreach contact do
        compute contact geometry: normal, penetration;
        if it is a real contact then
            find previous contact force;
            compute contact force subjected to the Coulomb condition;
            apply viscous damping if applicable;
            add contact force to the particle pair;
    // Integrating motion of all particles;
    foreach particle do
        compute resultant force;
        apply damping if applicable;
        update velocity and position;
        if (particle displacement > skin size ) set updateNL true;
```

Both cell-based and RT-based neighbor searching algorithms introduced in Section 2 are implemented to build the neighbor list. As for contact force computation and integration of particle motion, one can refer to our previous work, where a detailed description of GPU-DEM has been outlined for a 2D thread-block-wise code GODEM.[30] All algorithms depicted in this work are implemented in our in-house code CUDEM with CUDA C++ (11.2). The code is compiled on Linux/Ubuntu OS with OptiX 7.2.

4 | NUMERICAL EXAMPLES

4.1 | Simulation scenarios

We select three typical problems with different scales including granular packing, column collapse, and dry debris flow. All simulations have the same setting of material properties as listed in Table 2, and the gravitational acceleration and timestep are set to 10 m/s^2 and 0.001 s, respectively. The simulations are grouped into three different tests as outlined in Table 3.

As aforementioned, skin size influences the update frequency of neighbor list, which is investigated in Test 1. Problem size (typically particle number) and GPU specs may also have an effect on the computational performance, which are examined in Tests 2 and 3, respectively. We prepare four dedicated NVIDIA RTX GPU cards with specifications listed in Table 4, where RTX 3090 is the most powerful in the overall performance, and RTX 2080 Ti and RTX 5000 are the second and third. In Tests 1 and 2, RTX 5000 is utilized for a better benchmark. Detailed setups of the three tests are depicted in the following sections.

TABLE 2 Material properties in all simulations for the three problems.

Parameter	Value
Particle material density (kg/m^3)	2000
Particle contact stiffness k_n, k_t (N/m)	1×10^5
Wall contact stiffness k_n, k_t (N/m)	1×10^7
Particle coefficient of friction	0.5
Wall coefficient of friction	1.0
Viscous damping	0.5
Mass damping	0.1

TABLE 3 Overview of all simulation configurations for three tests.

Test group	Particles (M: million)	Skin size	RTX GPU
Test 1	0.8	0, 0.1, 0.2, 0.3, 0.4	5000
Test 2	0.1, 0.2, 0.4, 0.8, 1.6	0.2	5000
Test 3	1.6	0.2	3060, 5000, 2080Ti, 3090

TABLE 4 Specifications of testing GPUs.

NVIDIA GPU	RT cores	CUDA cores	Memory (GDDR6) bus/size
GeForce RTX 3060	28	3584	192 bit/12 GB
Quadro RTX 5000	48	3072	256 bit/16 GB
GeForce RTX 2080Ti	68	4352	352 bit/11 GB
GeForce RTX 3090	82	10,496	384 bit/24 GB

4.2 | Test 1: Column collapses with varying skin size

Column collapse test is selected to test the influence of skin size on the overall computational performance for both cell- and RT-based neighbor search algorithms. A total number of about 0.8 M monosized spheres of radius 0.1 m are setup as a $71 \times 80 \times 141$ lattice grid with an interval of 2.4 times of sphere radius, as shown in Figure 6A. To avoid particle stacking onto each other and forming a lattice configuration, all particles are shifted from the grid nodes with a random distance ranging from 0 to $0.1r$ in x and y directions. Then, particles are allowed to settle down under gravity until forming a stable heap in 40 s (i.e., 40,000 time steps), referring to Figure 6B for the final state. As can be seen, the granular column collapses to a relatively straight slope with a curved tail as observed in previous numerical and experimental studies, for example, Reference 36, and the angle of repose α is 27.4° that is close to the reported value of 28.1° in Reference 37, suggesting that our GPU-DEM code has been well implemented. Note that this work focuses on the acceleration scheme and the corresponding computational performance, and further validation and verification of the code are beyond the scope of this work.

Five groups of simulations are conducted on the Quadro RTX 5000 GPU card with a series of skin size factors 0, 0.1, 0.2, 0.3, and 0.4, respectively, where a larger skin size factor implies less frequent neighbor search in simulations. Notably, for the extreme case with zero skin size, the neighbor list needs updating every time step, which is generally time-consuming and not adopted in practical simulations. To measure the simulation performance at a given time instant, average speed is introduced as the accumulative time steps N_{tot} divided by the current elapsed time t_{ela}, that is,

$$\text{speed} = \frac{N_{\text{tot}}}{t_{\text{ela}}}. \tag{9}$$

Hence, a dynamic speed can be recorded during the simulation. As shown in Figure 7A, simulation speed drops rapidly at the very beginning of the simulation, which is attributed to the rapid increase of coordination number (number of

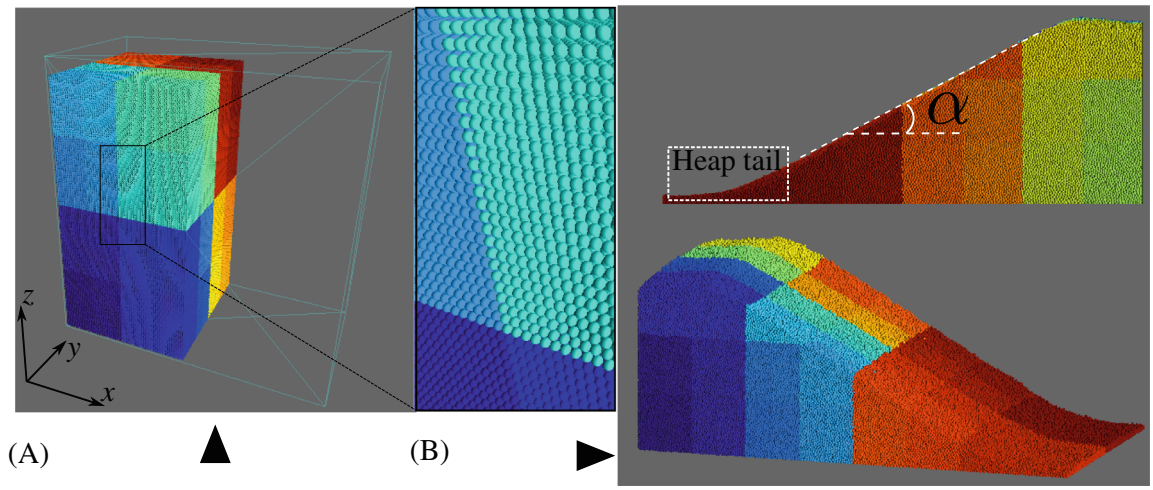

FIGURE 6 Simulation snapshots for column collapse of 0.8 M particles. (A) Initial state; (B) final state

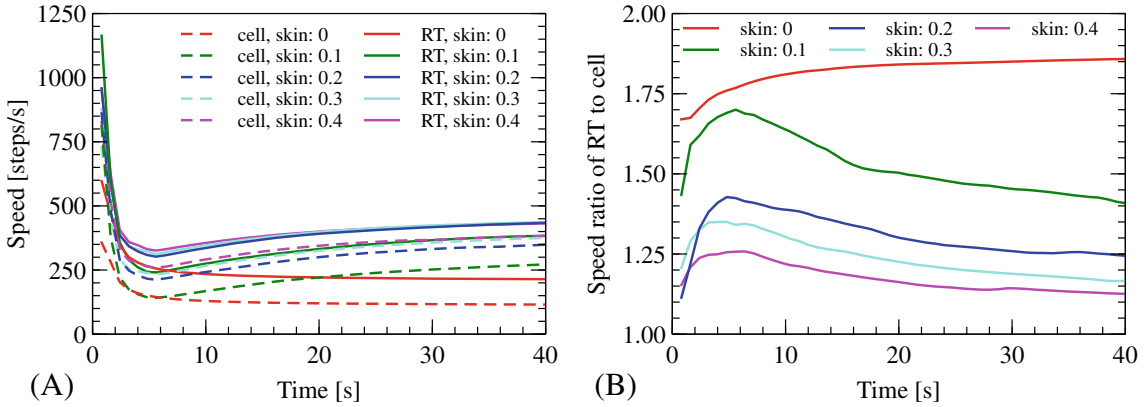

FIGURE 7 Evolution of average speed with time for column collapses with 0.8 M particles and varying skin sizes on the Quadro RTX 5000 GPU.

contacts per particle) during column collapse. Then, simulation speed is expected to converge to a relatively stable value for each simulation setup when the heap approaches to a static state. Moreover, it is not surprising that both initial and stable simulation speeds increase with skin size. This is because a larger skin size is helpful to cache more neighbors among which some may be not touching with the given particle, thereby reducing the update frequency of neighbor list. Nevertheless, the overall speed cannot be improved furthermore when skin size factor gets beyond 0.2. Therefore, a skin size factor of 0.2 is selected for the rest simulations of this work. Moreover, it is worth pointing out that increasing skin size might result in a side effect of a potential increase of memory allocation. In this work, we have a fixed neighbor number of 32 for each particle. That is to say, the neighbor list can only cache up to 32 neighbors for each particle. One has to increase the memory of neighbor list when more neighbors need to be cached.

The analysis above is applicable to both cell- and RT-based neighbor search algorithms. Also, for each skin size, the RT-based simulation has a higher speed than the cell-based one, indicating that we can gain improved performance when using the RT-based neighbor search algorithm. Quantitatively, the speed ratio of RT-based approach to cell-based approach is plotted in Figure 7B. It can be seen that the speed ratio increases rapidly first, then gradually drops except for the case without skin size. For the case without skin size, the speed ratio increases monotonically, up to 1.9, which means using the RT core can be up to 1.9× faster when we have to update the neighbor list every time step. It implies that we can gain better performance with the acceleration of RT core when maximizing its usage. Indeed, the usage of RT core is likely to be reduced when increasing skin size because the update frequency of neighbor list is reduced. It also suggests that the RT-based algorithm is suitable for dynamic simulations, referring to the rapid increase

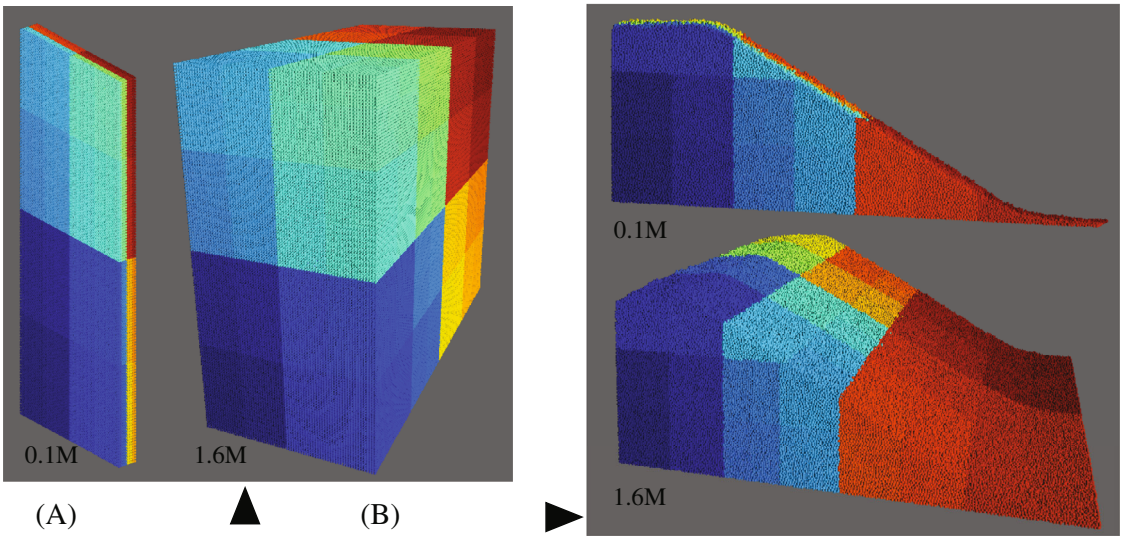

FIGURE 8 (A) Initial and (B) final states for simulations of column collapse with 0.1 and 1.6 M particles.

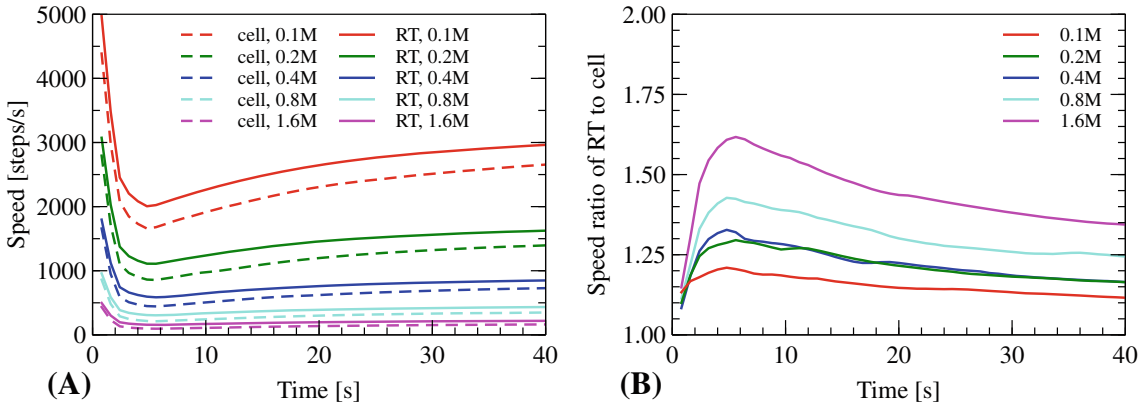

FIGURE 9 Evolution of average speed for column collapses with varying particle numbers on the Quadro RTX 5000 GPU.

of speed ratio in the beginning few hundreds of time steps, during when the particles rearrange dramatically due to collapsing.

To sum up, in the present test, the stable simulation speed can be up to 450 steps per second for 0.8 M particles on the Quadra RTX 5000 GPU card, and using the RT core can achieve a speed up of 1.2–1.4.

4.3 | Test 2: Column collapses with varying particle number

Similar to the setup in Test 1, column collapse simulations are carried out with different particle numbers ranging from 0.1 to 1.6 M. To have a better comparison, only the y dimension of the column is adjusted to prepare different particle numbers as shown in Figure 8.

Figure 9 shows the average speed and speed ratio between the RT-based and cell-based algorithms for different particle numbers during column collapse simulations. With particle number increasing from 0.1 to 1.6 M, the initial simulation speed drops significantly from 5000 to 450 steps per second, while the stable simulation speed ranges from 3000 to 220 steps per second. For all cases, the overall performance with the RT-based algorithm is better than that with the cell-based one. Moreover, with particle number increasing, the RT core can be utilized more extensively with an increase speed ratio. In particular, for the case with 1.6 M particles, the speed ratio can reach 1.6 and 1.3 at the peak and final, respectively.

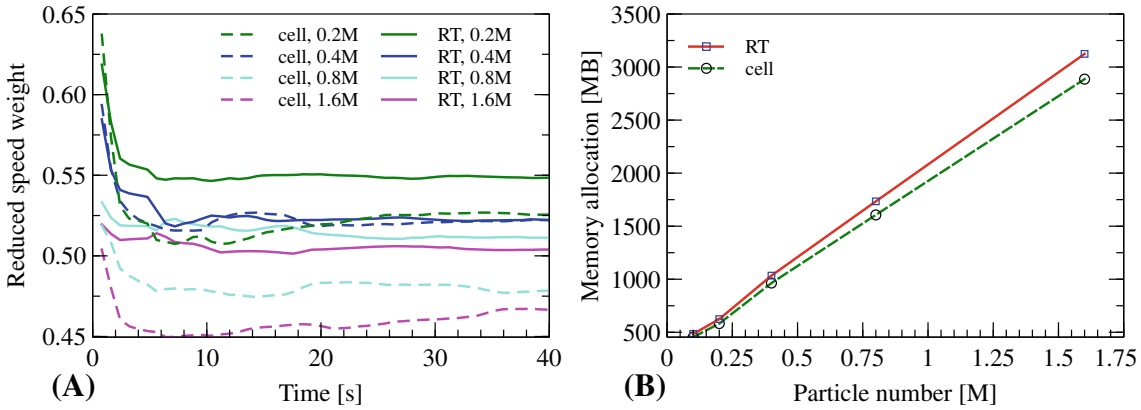

FIGURE 10 (A) Evolution of reduced speed weight, (B) memory allocation for column collapse with varying particle numbers.

To compare the scalability of the cell- and RT-based algorithms, we introduce a quantity coined as reduced speed weight, defined as the ratio of simulation speed for the case with 2N particles to that with N particles. As the name suggests, reduced speed weight shows how much the overall simulation speed becomes when doubling particle number. For reduced speed weight equal to 0.5, it suggests that the system can be linearly scaled without degradation of simulation performance. For a system with reduced speed weight greater than 0.5, it means that the hardware has not been maximally utilized, otherwise there is shortage in hardware for reduced speed weight less than 0.5. As shown in Figure 10A, a larger particle system has a smaller reduced speed weight as expected. In particular, the RT-based simulations have greater reduced speed weight than the cell-based one for all cases, indicating that the RT-based algorithm can be scaled more efficiently. Note that the hardware here denotes all hardware resources in the computing system such as CUDA cores, RT cores, GPU memory size and bus. An elaborate investigation of these resources is complicated and also beyond the scope of this work. Only memory allocation is shown here in Figure 10B, where the memory allocation is observed to increase almost linearly with increasing particle number. Besides, the RT-based simulation requires more but not too much memory than the cell-based one.

4.4 | Test 3: Different GPU specs

In addition to the column collapse simulation in Tests 1 and 2, two more problems, namely granular packing and dry debris flow, are simulated in Test 3. On the top of the findings and discussions in Test 2, 1.6 M particles are used for all the simulations of Test 3.

4.4.1 | Granular packing and column collapse

The granular packing simulation has almost the same setup as the column collapse except an additional side wall at the right surface of the column. Figure 11 shows a sequence of snapshots during simulation of granular packing with 1.6 M particles.

Figure 12 shows the evolution of simulation speed for packing and column collapse tests with 1.6 M particles on different GPUs. It can be seen that both packing and column collapse have almost the same simulation speed at the very beginning (before around 2000 time steps). This is because most of the particles just settle down at this period for both cases. Moreover, the RT-based simulation is faster than the cell-based one for both packing and column collapse problem across all GPUs. Specifically, the simulation speed is positively proportional to the overall power of the GPU as expected in Figure 13A. Interestingly, the packing simulation is faster than the column collapse one, whereas the column collapse simulation has a larger RT-cell speed ratio than the packing one as shown in Figure 13B. The main reason is that the column collapse simulation is more dynamic due to the large-scale rearrangements of particles, which implies more frequent updates of the neighbor list. It also verifies that maximally utilizing the RT core can give a better performance as mentioned in Test 1. Furthermore, it is worth noting that the performance gain (in terms of speed ratio) of the RT-based approach is determined by the weight of the RT core in the overall performance of a GPU but not just the number of RT cores. For example, RTX 2080Ti has the most RT cores, but its speed ratio is the lowest.

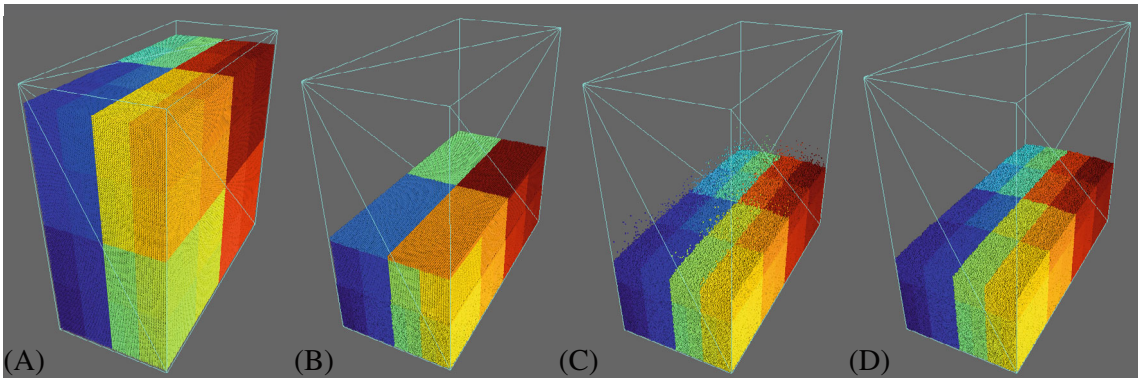

FIGURE 11 Simulation snapshots for granular packing with 1.6 M particles. (A) $t = 0$ s; (B) $t = 2$ s; (C) $t = 4$ s; (D) $t = 40$ s

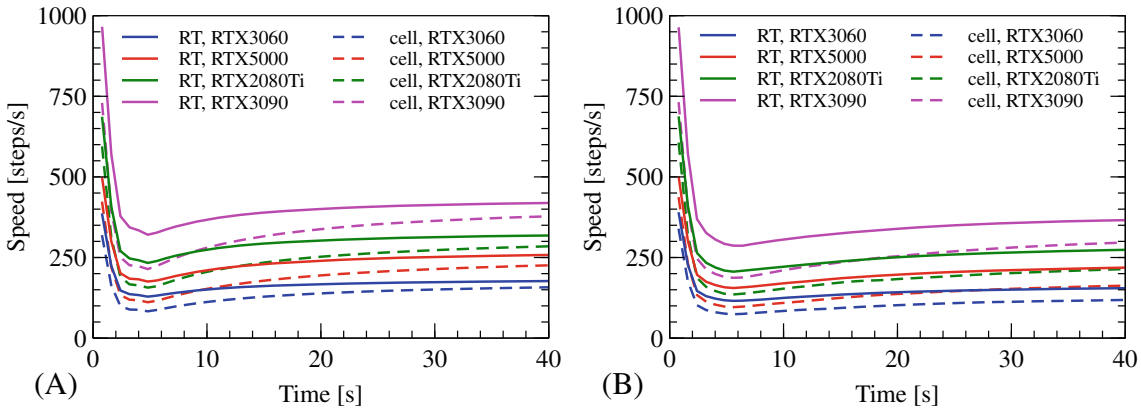

FIGURE 12 Evolution of speed for (A) packing and (B) column collapse simulations on different GPUs (1.6 M particles).

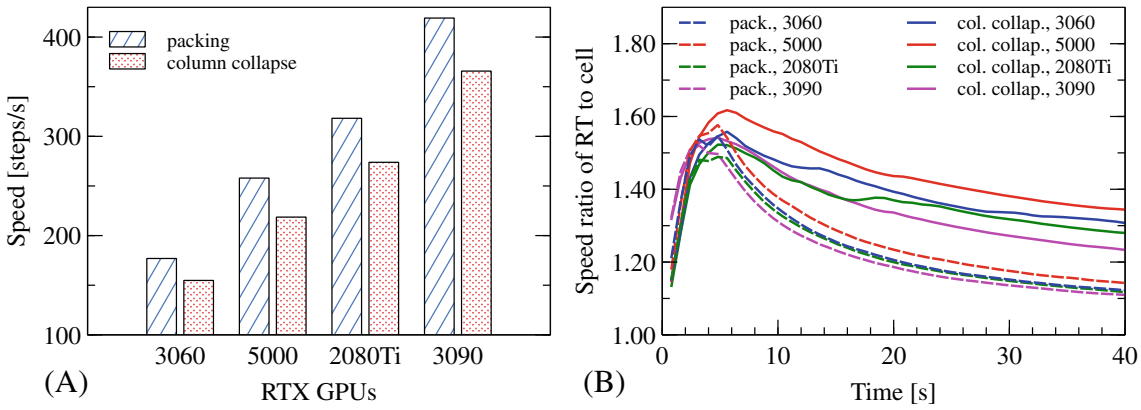

FIGURE 13 Packing and column collapse simulations on different GPUs (1.6 M particles): (A) The final simulation speed with the RT-based algorithm and (B) evolution of speed ratio.

4.4.2 | Dry debris flow

As for the dry debris flow, the simulation setup is more complex as it involves a large-scale of moving debris particles and a static highly irregular-shaped terrain boundary, as shown in Figure 14. In this work, the terrain is modeled by a triangle mesh, which is created based on a digital elevation model. The debris particles are generated from a lattice grid of the slide region, that is, pre-slide terrain subtracted by the after-slide terrain. The lattice grid has a spacing of 0.25 m, which results in an amount of about 1.6 M debris particles with radii of 0.125 m. Note that neighboring triangle facets

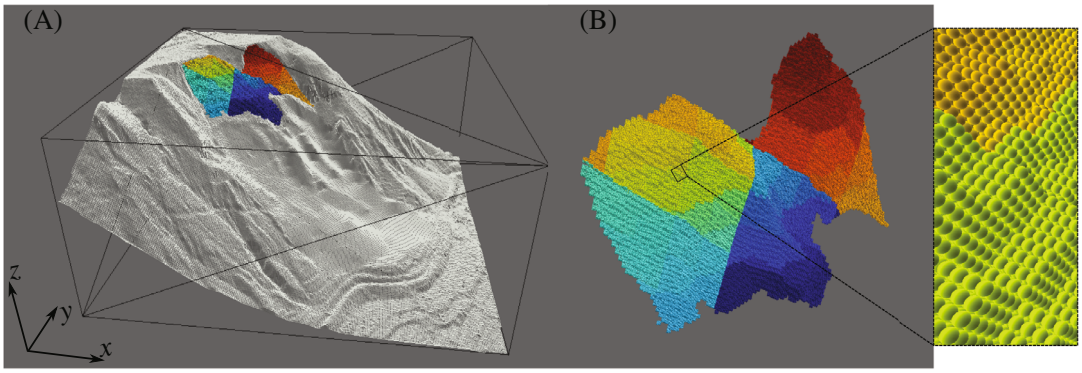

FIGURE 14 Simulation setup for dry debris flow: (A) terrain and sliding source, and (B) zoom-in sliding source.

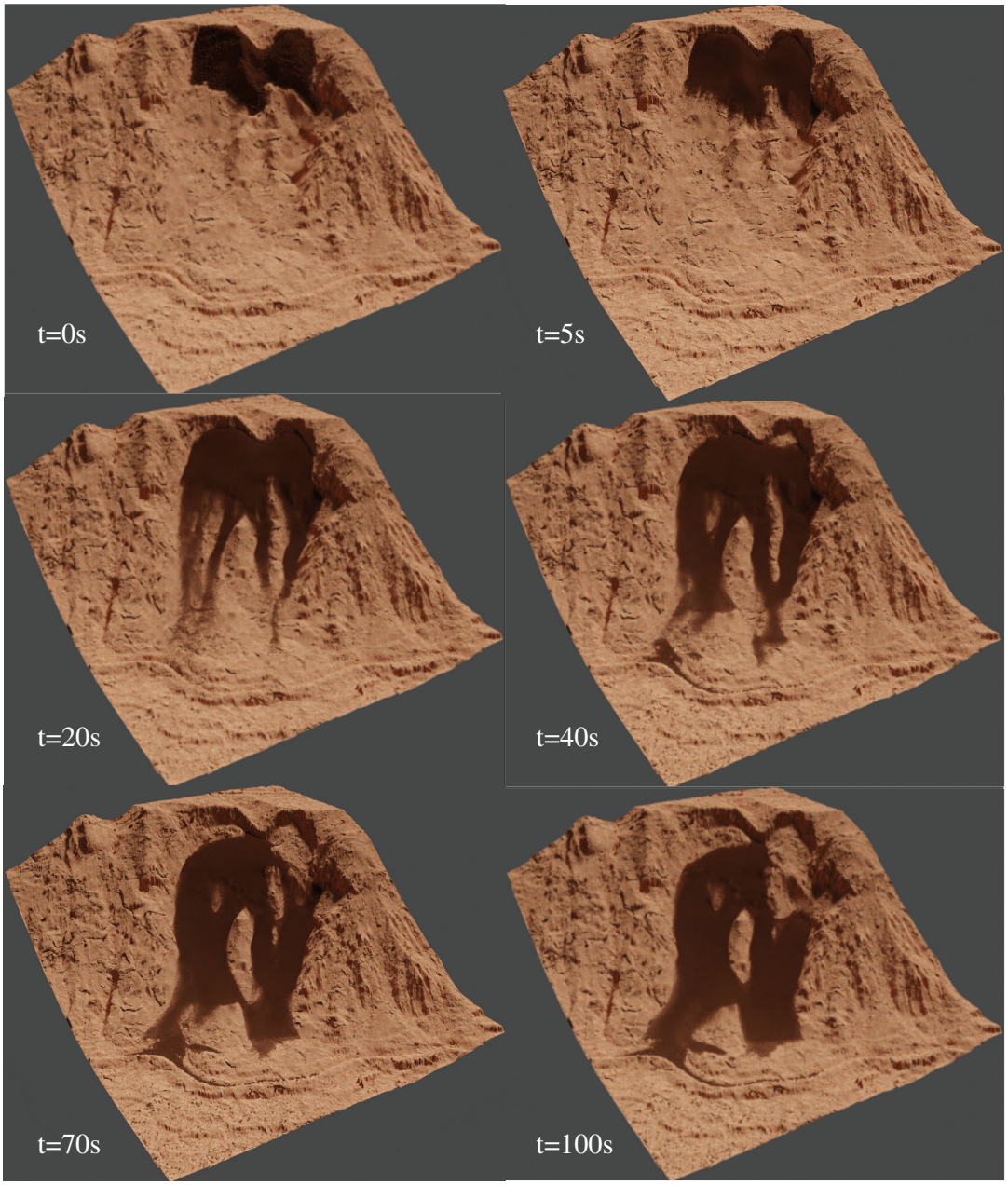

FIGURE 15 Simulation snapshots for dry debris flow.

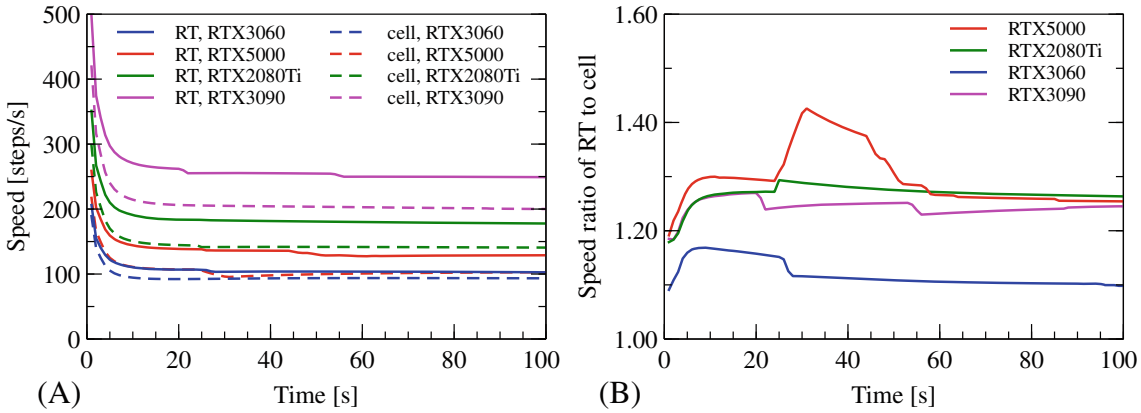

FIGURE 16 Average simulation speed for the dry debris flow on different GPUs (1.6 M particles).

of particles are not cached for efficiency, since the neighboring facets can be efficiently identified with the position of a given particle due to the regular mesh of a digital elevation model. In case a particle is contact with multiple triangles, the contact force is weighted by the particle-triangle contact area for accuracy.

The dry debris flow is simulated for 100,000 time steps with a fixed timestep of 0.001 s. Figure 15 shows several snapshots (rendered in blender) of the debris flow at time 0, 5, 20, 40, 70, and 100 s, respectively. The simulation captures reasonably well the profiles and evolution of general dry debris flow. The simulation speed and speed ratio of the debris flow simulation are recorded in Figure 16. Similar to the column collapse case, the simulation speed drops dramatically during the first several hundreds of time steps, and it plateaus afterwards. The speed ratio of the RT-based approach to cell-based one during the whole simulation varies between 1.1 and 1.6, depending on the GPU specs. For the memory consumption, it takes about 3138 MB for the RT-based simulation, slightly lower than that of the cell-based one (3418 MB). However, for the column collapse problem, the RT-based simulation requires slightly more memory than that of the cell-based one, as shown in Figure 10B. It suggests that both RT-based and cell-based simulations have similar memory consumption. Overall, the RT-based approach exhibits better computational performance (in terms of computational efficiency and memory consumption) than the cell-based one.

5 | SUMMARY

This work presented an RT-based neighbor search framework for particle-based numerical methods including DEM, MD, PD, and SPH on NVIDIA's RTX GPUs. In the proposed framework, the neighbor search problem is converted to a ray-tracing one so that it can leverage the specially-designed RT cores for hardware-wise acceleration. Specifically, a particle is considered as a neighbor of a given particle if an infinitesimally short ray starting from the particle hits the AABB of the given particle. The implementation based on the CUDA and OptiX APIs are presented, where the state-of-the-art BVH algorithm is employed via OptiX. We have discussed the use of skin to reduce the frequency of neighbor search, prior to applying in large-scale DEM simulations as a showcase. Numerical tests, including granular packing, column collapse, and dry debris flow, are performed to evaluate the computational performance of the proposed RT-based approach against the conventional cell-based one. The results indicate that the RT-based simulations can be approximately 10%–60% faster than the cell-based ones, requiring almost as much memory as the cell-based one. The performance gain could be more notable for large particle systems and dynamic problems, where the RT cores can be utilized to the most extent.

Note that the current release of RT drivers and APIs are specially designed for computer graphics and yet not optimized for computational mechanics. This work serves as an effort on leveraging the emerging RT cores and techniques for computational mechanics. It is anticipated that, with the improvements of RT drivers, the RT cores would be another promising and powerful plus resource complementary to CUDA cores for parallel computing in computational mechanics. It is also worth noting that only the traversal unit of RT cores is utilized in this work, while the applicability of ray-triangle intersection unit merits further exploration.

ACKNOWLEDGMENTS

This work was financially supported by the National Natural Science Foundation of China (51909095, 51909289, and 11972030), the Guangdong Basic and Applied Basic Research Foundation (2022A1515010848), Research Grants Council of Hong Kong (16211221), and the Project of Hetao Shenzhen-Hong Kong Science and Technology Innovation Cooperation Zone (HZQB-KCZYB-2020083). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the financial bodies.

DATA AVAILABILITY STATEMENT

The data and source code that support the findings of this study are available from the authors upon reasonable request.

ORCID

Shiwei Zhao https://orcid.org/0000-0002-3410-3935
Zhengshou Lai https://orcid.org/0000-0002-2378-9193
Jidong Zhao https://orcid.org/0000-0002-6344-638X

REFERENCES

1. Cundall P, Strack O. A discrete numerical model for granular assemblies. *Géotechnique*. 1979;29(1):47-65.
2. Frenkel D, Smit B. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press; 2002.
3. Silling SA. Reformulation of elasticity theory for discontinuities and long-range forces. *J Mech Phys Solids*. 2000;48(1):175-209.
4. Gingold RA, Monaghan JJ. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Not Royal Astron Soc*. 1977;181(3):375-389.
5. O'Sullivan C. *Particulate Discrete Element Modelling: A Geomechanics Perspective*. CRC Press; 2011.
6. Zhao S, Zhao J. A poly-superellipsoid-based approach on particle morphology for DEM modeling of granular media. *Int J Numer Anal Methods Geomech*. 2019;43(13):2147-2169.
7. Coming DS, Staadt OG. Kinetic sweep and prune for collision detection. *Comput Graph*. 2006;30(3):439-449.
8. Tracy DJ, Buss SR, Woods BM. Efficient large-scale sweep and prune methods with AABB insertion and removal. Proceedings of the 2009 IEEE Virtual Reality Conference; 2009:191-198; IEEE.
9. Kozicki J, Donze FV. A new open-source software developed for numerical simulations using discrete modeling methods. *Comput Methods Appl Mech Eng*. 2008;197(49-50):4429-4443.
10. Zhao S, Zhao J. SudoDEM: unleashing the predictive power of the discrete element method on simulation for non-spherical granular particles. *Comput Phys Commun*. 2021;259:107670.
11. Tian Y, Zhang S, Lin P, Yang Q, Yang G, Yang L. Implementing discrete element method for large-scale simulation of particles on multiple GPUs. *Comput Chem Eng*. 2017;104:231-240.
12. Lubbe R, Xu W-J, Wilke DN, Pizette P, Govender N. Analysis of parallel spatial partitioning algorithms for GPU based DEM. *Comput Geotechn*. 2020;125:103708.
13. Xu J, Qi H, Fang X, et al. Quasi-real-time simulation of rotating drum using discrete element method with parallel GPU computing. *Particuology*. 2011;9(4):446-450.
14. Gan JQ, Zhou ZY, Yu AB. A GPU-based DEM approach for modelling of particulate systems. *Powder Technol*. 2016;301:1172-1182.
15. Govender N, Wilke DN, Kok S. Blaze-DEMGPU: modular high performance DEM framework for the GPU architecture. *SoftwareX*. 2016;5:62-66.
16. Spellings M, Marson RL, Anderson JA, Glotzer SC. GPU accelerated Discrete Element Method (DEM) molecular dynamics for conservative, faceted particle simulations. *J Comput Phys*. 2017;334:460-467.
17. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J Comput Phys*. 2008;227(10):5342-5359.
18. Le Grand S, Götz AW, Walker RC. SPFP: speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. *Comput Phys Commun*. 2013;184(2):374-380.
19. Phillips JC, Hardy DJ, Maia JD, et al. Scalable molecular dynamics on CPU and GPU architectures with NAMD. *J Chem Phys*. 2020;153(4):044130.
20. Diehl P. *Implementierung eines peridynamik-verfahrens auf gpu*, Master's thesis; 2012.
21. Liu W, Hong J-W. Discretized peridynamics for brittle and ductile solids. *Int J Numer Methods Eng*. 2012;89(8):1028-1046.
22. Mossaiby F, Shojaei A, Zaccariotto M, Galvanetto U. OpenCL implementation of a high performance 3D peridynamic model on graphics accelerators. *Comput Math Appl*. 2017;74(8):1856-1870.
23. Harada T, Koshizuka S, Kawaguchi Y. Smoothed particle hydrodynamics on GPUs. *Computer Graphics International*. Vol 40. SBC Petropolis; 2007:63-70.
24. Domínguez JM, Crespo AJ, Gómez-Gesteira M. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. *Comput Phys Commun*. 2013;184(3):617–627.
25. NVIDIA. NVIDIA turing GPU architecture: graphics reinvented; 2018. https://bit.ly/2NGLr5t

26. Wald I, Usher W, Morrical N, Lediaev L, Pascucci V. RTX beyond ray tracing: exploring the use of hardware ray tracing cores for tet-mesh point location. *High Perform Graph (Short Pap)*. 2019;7-13.
27. Wang B, Wald I, Morrical N, et al. An GPU-accelerated particle tracking method for Eulerian–Lagrangian simulations using hardware ray tracing cores. *Comput Phys Commun*. 2022;271:108221.
28. Zellmann S, Weier M, Wald I. Accelerating force-directed graph drawing with rt cores. Proceedings of the 2020 IEEE Visualization Conference (VIS); 2020:96-100; IEEE.
29. Evangelou I, Papaioannou G, Vardis K, Vasilakis A. Fast radius search exploiting ray-tracing frameworks. *J Comput Graph Techn*. 2021;10(1):25-48.
30. Zhao S, Zhao J, Liang W. A thread-block-wise computational framework for large-scale hierarchical continuum-discrete modeling of granular media. *Int J Numer Methods Eng*. 2021;122(2):579-608.
31. Karras T, Aila T. Fast parallel construction of high-quality bounding volume hierarchies. Proceedings of the 5th High-Performance Graphics Conference; 2013:89-99.
32. Ströter D, Mueller-Roemer JS, Stork A, Fellner DW. OLBVH: octree linear bounding volume hierarchy for volumetric meshes. *Vis Comput*. 2020;36(10):2327-2340.
33. Parker SG, Bigler J, Dietrich A, et al. Optix: a general purpose ray tracing engine. *ACM Trans Graph (TOG)*. 2010;29(4):1-13.
34. Morton GM. A computer oriented geodetic data base and a new technique in file sequencing.
35. Moon B, Jagadish HV, Faloutsos C, Saltz JH. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans Knowl Data Eng*. 2001;13(1):124-141.
36. Alonso J, Herrmann H. Shape of the tail of a two-dimensional sandpile. *Phys Rev Lett*. 1996;76(26):4911.
37. Zhou Y, Xu B, Yu A, Zulli P. Numerical investigation of the angle of repose of monosized spheres. *Physical Review E*. 2001;64(2):021301.

How to cite this article: Zhao S, Lai Z, Zhao J. Leveraging ray tracing cores for particle-based simulations on GPUs. *Int J Numer Methods Eng*. 2023;124(3):696-713. doi: 10.1002/nme.7139