Engineering Analysis with Boundary Elements 172 (2025) 106133

Contents lists available at ScienceDirect

Engineering Analysis with Boundary Elements

journal homepage: www.elsevier.com/locate/enganabound

A fast bond-based peridynamic program based on GPU parallel computing

Yang Yang [a,*], Zixin Su [b], Yijun Liu [b]

[a] Faculty of Material Science, Shenzhen MSU-BIT University, Guangdong
[b] Department of Mechanics and Aerospace, Southern University of Science and Technology, Guangdong

ARTICLE INFO

Key words:
Bond-based peridynamic
GPU parallel
CUDA programming technology

ABSTRACT

Peridynamic is an effective method for addressing fracture problems. However, the non-local theory makes it time-consuming. Although some techniques have been developed to improve computational efficiency, the acceleration effect remains relatively limited. This paper introduces a parallel algorithm for bond-based peridynamic using the GPU parallel CUDA programming technology. The calculation process is divided into functions with material points and bonds as the smallest calculation units. The loop of material points and bonds is changed to the index to achieve parallelism. A general horizon generation module is established to optimize storage. Additionally, a general register technique is proposed for high-speed access register memory to reduce global memory access. This technique not only eliminates the restriction on the number of horizon points, also suitable for nonuniform distribution of material points. Compared to serial and OpenMP parallel programs, the present algorithm can achieve up to 800-fold and 100-fold acceleration, respectively. In a typical simulation of one million particles, executing 4000 iterations can be completed in 5 minutes for single precision and 20 minutes for double precision on a low-end GPU PC.

1. Introduction

Peridynamic (PD) [1] is a non-local theoretical framework. This theory solves the problem of modeling discontinuous space by introducing integral based governing equations. This approach makes it more useful for modeling crack propagation without altering the mesh. In the implementation, the domain needs to be discretized into a series of material points, each containing information about position, volume, and density. The time complexity of PD is O(*PN*), where *P* is the total number of material points and *N* is the number of horizon points for each material point. To ensure the stability and accuracy of numerical computations, a small incremental step must be set, which leads to a large number of iterations requiring substantial computational resources. At this scale, PD requires more computational effort and larger storage space compared to classical continuum mechanics based methods, such as finite element method (FEM) resulting in lower computational efficiency.

In order to improve the computational efficiency of PD, researchers have proposed a range of related methods. Initially, mathematical optimization and algorithmic improvements are utilized to theoretically reduce the amount of computation [2,3]. Jafarzadeh and colleagues [4] employed convolution and Fast Fourier Transform to decrease the problem complexity from O(MN) to O(NlogN) [5], and applied this method in MATLAB to create a peridynamic analysis program named PeriFast [6]. Tao Ni et al. [7] proposed an ordinary state based peridynamic (OSBPD) model based on matrix operation for a rapid solution, effectively reducing the computation involved in common loop forms. On the other hand, local and nonlocal coupling [8,9], as well as refined meshing [10,11], are also effective ways to reduce the computational load of PD. Local and nonlocal coupling [12,13] is typically achieved by integrating PD with traditional continuous medium mechanics [14,15]. Refined meshing [16] essentially involves sparsely distributing points in areas without cracks or damage, while densely distributing points in areas with cracks. These methods can reduce the computational load of PD to a certain extent, but they introduce new issues, such as the need to know in advance the locations of cracks, damage, etc. [17,18], and the need for manual parameter tuning, which may affect the universality of PD. Moreover, the refined meshing method may also require the implementation of adaptive meshing functions, increasing the difficulty of coding. Additionally, some studies have used machine learning [19,20] and deep learning [21,22] to accelerate the computation process of PD, but the acceleration effect is relatively limited.

Parallel computing is an effective approach for accelerating numerical computations, which is mainly divided into Central Processing Unit

* Corresponding author.
E-mail address: yangy2023@smbu.edu.cn (Y. Yang).

https://doi.org/10.1016/j.enganabound.2025.106133
Received 24 November 2024; Received in revised form 3 January 2025; Accepted 19 January 2025
Available online 29 January 2025
0955-7997/© 2025 Published by Elsevier Ltd.

Table 1
Functions of implementation process.

Executed times	BBPD	BBPD with Cracks
Once	Structural discretization Neighborhood point generation Boundary conditions	Structural discretization Neighborhood point generation Boundary conditions Initial crack setting
Multi-times	Internal forces calculation Updating displacements	Internal forces calculation Updating displacements Crack judgement Damage judgement

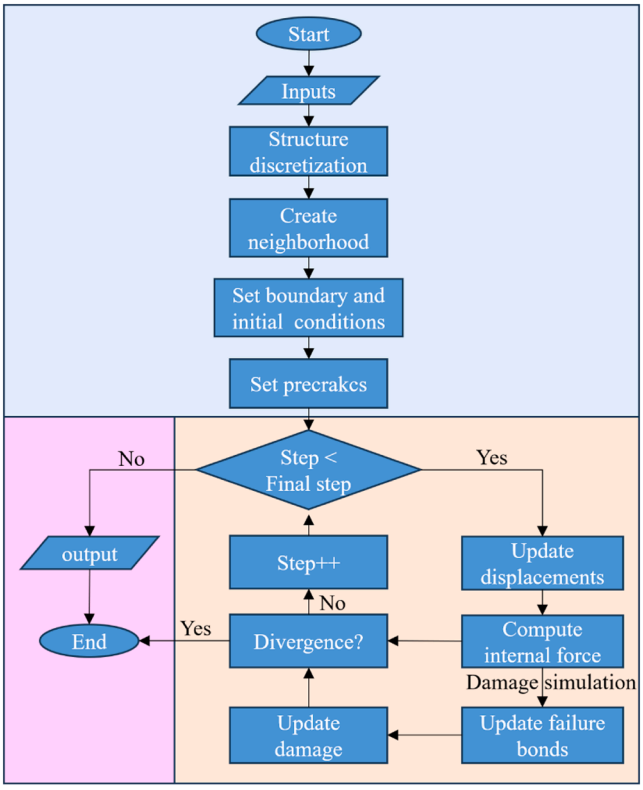

Fig. 1. The flowchart of BBPD.

(CPU) based parallelism and Graphic Processing Units (GPU) based parallelism [23]. CPU parallelism is more suitable for situations with complex logic, such as Message Passing Interface (MPI), Shared Memory Parallel Programming (Open Multi-Processing), etc. On the other hand, GPU parallelism is more suitable for situations with simple logic but large computational volumes [24], such as Open Computing Language (OpenCL), NVIDIA's Compute Unified Device Architecture (CUDA), etc. Due to the non-local nature of Peridynamics, each material point only interacts with points within its own neighborhood, making it highly suitable for parallelization [25].

Sun [26] developed a simple peridynamic framework on a heterogeneous computing platform of CPUs and GPUs using OpenMP and CUDA, respectively. Liu et al. [27] implemented the parallelization of PD on GPU with CUDA technology, achieving a speedup of 2.6 to 10.3 times compared to CPU serial program. Li et al. [28] transplanted and optimized Peridigm on SIMT accelerators. They achieved a maximum performance of 825.72 TFLOPS on four Nvidia Tesla V100 GPUs, which is 10.24 times faster than the original Peridigm, and reached a maximum scale of 36,096,000 material points. Zhong et al. [29] coupled peridynamics with the finite element method, proposing a parallel optimization framework named PeriFEM. Compared to the FEM method, they achieved similar computational rates of the same scale. Mossaiby et al. [30] developed a high-performance PD computing model using OpenCL, which showed a speedup of 3 to 6 times compared to the OpenMP parallel code in 3D tests with hundreds of thousands to millions of nodes, and expanded the maximum calculation scale to 3550,261 material points. Boys et al. [31] developed a Python code library named PeriPy based on bond based peridynamic (BBPD) using OpenCL. This software achieved a scale of tens of millions of nodes on Geforce RTX 2080Ti, with a 1.4 to 2.0 times improvement over OpenCL and a 3.7 to 7.3 times improvement over OpenMP. Bartlett et al. [32] expanded the program to state based peridyanmic (SBPD) based on PeriPy, expanded the maximum problem scale to 300 million, and increased the speed by 63 times compared to linear programs. Wang et al. [33] used warp functions on CUDA, by limiting the number of neighboring points in the model to achieve hundreds of times the acceleration efficiency compared to serial programs.

Most of the acceleration optimizations in CPU parallel acceleration have been targeted at examples with less than 1 million points. For large-scale simulations, it is usually necessary to rely on supercomputers or multi-CPU structures, which require higher equipment specifications and are not user-friendly. Research on parallel of peridynamics based on GPU has mostly been limited to transforming serial programs into parallel ones. Many optimization strategies bring acceleration effects that mainly depend on the improvement of the GPU's own performance, while optimizations designed to fit the hardware structure of the GPU are relatively limited.

In addition, there are still some issues in GPU parallel computing:

Algorithm 1
Parallel pseudocode of points based function.

```
1 Function Serials Program:
2     for i ← 0 to nt do
3         Calculation;
4     end
5 Function Parallel Program;
6     nodeIdx ← threadIdx.x + blockIdx.x × blockDim.x;
7     if nodeIdx < nt then
8         Calculation;
9     end
```

Algorithm 2
Parallel pseudocode for bond based function.

```
Data: gpuIdx, threadIdx.x, blockIdx.x, blockDim.x, maxNeighbor, nodeIdx, targetIdx,
      Rex, nt, numfam[]
Result: res
gpuIdx ← threadIdx.x + blockIdx.x * blockDim.x;
nodeIdx ← gpuIdx / maxNeighbor;
targetIdx ← gpuIdx % maxNeighbor;
res ← 0.0;
if nodeIdx < nt and targetIdx < numfam [nodeIdx] then
    res ← Calculation();
    atomicAdd(&target[nodeIdx], res);
end
```

(1) The memory space allocated for storing neighborhood points does not have a predetermined size, which leads to the inefficient use of thread and memory resources. This results in a wastage of memory and computational resources, making it challenging for GPUs to handle large-scale problems.
(2) Most GPU parallel calculations still heavily rely on global memory and have not fully utilized CUDA's memory structure, resulting in a waste of memory bandwidth.
(3) Most PD parallel algorithm lacks general utility. Some may restrict the size of the neighborhood, handle only uniformly distributed and undamaged discrete structures, or limit the theory of PD.

Based on the limitations mentioned above, a bond-based general-PD parallel algorithm on CUDA based on GPU parallelism is developed to improve memory usage and computational efficiency. The algorithm implements a particle parallel mode, as well as a more efficient accessing and storage strategy using register bandwidth. This eliminates the limitation of the number of neighborhood points and enhances the search speed, resulting in significant speedups compared to the serial program and other parallel algorithms. This allows for quick analysis of the deformation and crack propagation in BBPD and the framework is also applicable to other PD theories.

The rest of this paper is organized as follows: Section 2 introduces the bond-based PD theory and the parallel strategy. The general PD program is validated through numerical analyses in Section 3. The performance of the developed parallel program, its generalization ability, and efficiency are discussed in Section 4. Section 5 presents the conclusions.

2. Methodology

2.1. Bond-based PD theory

Bond-based PD was first introduced in 2000 [1] and is widely applied. Its motion equation can be expressed as follows:

$$\rho\ddot{\mathbf{u}}(\mathbf{x},t)=\int_{H_{\mathbf{x}}}\mathbf{f}[\mathbf{u}'-\mathbf{u},\mathbf{x}'-\mathbf{x},t]dV_{\mathbf{x}'}+\mathbf{b}(\mathbf{x},t), \tag{1}$$

where $\mathbf{u}$, ρ, $\mathbf{b}$ and $V_{\mathbf{x}'}$ are the displacement, density, body force per unit volume at material point $\mathbf{x}$ and the volume of material point $\mathbf{x}'$. The neighborhood $H_{\mathbf{x}}$ is called horizon, which is usually taken to be a spherical with radius δ. $\mathbf{f}$ represents the pairwise force density vector, which is computed by:

$$\mathbf{f}(\eta,\xi)=\frac{\eta+\xi}{|\eta+\xi|}cs\mu. \tag{2}$$

$\xi=\mathbf{x}'-\mathbf{x}$ is the initial relative position, $\eta=\mathbf{u}'-\mathbf{u}'$ is the current relative displacement. c is the micromodule. s represents the stretch of a bond:

$$s=\frac{|\eta+\xi|-|\xi|}{|\xi|}. \tag{3}$$

The force density vector can be modified through a history-dependent scalar valued function μ as Eq. (4). When the stretch s of a bond exceeds its critical stretch s_c, failure occurs.

$$\mu(\xi,t)=\begin{cases}1,\text{if}s(\xi,t')<s_c\text{forall}0\le t'\le t\\ 0,\text{otherwise}\end{cases}. \tag{4}$$

Local damage at a point is defined as the weighted ratio of the number of eliminated interactions to the total number of initial interactions of a material point with its family members. The local damage at a point can be quantified as

$$\varphi(\mathbf{x},t)=1-\frac{\int_{H_{\mathbf{x}}}\mu(\xi,t)dV_{\mathbf{x}'}}{\int_{H_{\mathbf{x}}}dV_{\mathbf{x}'}}. \tag{5}$$

2.2. Parallel strategy

The implementation process of BBPD algorithm is outlined in Table 1 and Fig. 1. The steps involve structural discretization, neighborhood point generation, and the application of boundary conditions, which are executed only once. Throughout the time iteration process, functions such as calculating internal forces and updating displacements require thousands or even tens of thousands of iterations, making it the most time-consuming part. Consequently, optimizing the time-step loop functions is crucial.

In the BBPD analysis framework, functions can be broadly categorized into three types:

(1) Functions based on material points as the smallest computational units: These functions require information that is only related to the material points.
(2) Functions based on bonds as the smallest computational units: These functions are predominantly located within the time step loop. As a result, they consume a significant amount of time and

Algorithm 3
Neighborhood points generation.

```
Input: node[], nt, radius
Output: nodefam[]
Function Find-NeighborNum ( ) :
    nodeIdx ← threadIdx.x + blockIdx.x * blockDim.x;
    if nodeIdx < nt then
        for i:nt do
            if distance(node[nodeIdx], node[i]) < radius then
                numfam[nodeIdx] ++;
            end
        end
    end
Function Get-MaxNeighbors ( ) :
    nodeIdx ← threadIdx.x + blockIdx.x * blockDim.x;
    if nodeIdx < pdoConst._ntp then
        if numfam[nodeIdx] > maxNeighbor then
            maxNeighbor ← numfam[i];
        end
    end
    atomicMax(maxNeighbor, currMax);
Function Find-Neighbors ( ) :
    nodeIdx ← threadIdx.x + blockIdx.x * blockDim.x;
    if nodeIdx < nt then
        for i:nt do
            if distance(node[nodeIdx], node[i]) < radius then
                nodefam[numfam[nodeIdx]] ← i;
            end
        end
    end
```

spatial resources. Therefore, these functions will be the primary targets for optimization.

(3) Neighborhood search functions: This type of function is quite special. When performing a global search, it requires searching all bonds based on particles as the smallest computational units.

The above categorization will serve as the foundation for subsequent optimizations. The traditional BBPD program based on CPU serial computation serves as a baseline to compare with the subsequent optimized programs.

2.3. Parallel based on material points

The original loop for P material points is divided into P threads to execute the calculations within the loop in parallel. Algorithm 1 provides the pseudocode form for parallelizing computations related to points. At this stage, the functions associated with points have achieved the finest level of parallelism granularity. The global thread coordinates for the points are set as follows:

$$\text{nodeIdx} = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x}, \tag{6}$$

Where, threadId.x is the local index of the thread within its block, indicating the position of the current thread within the block. blockIdx.x is the index of the block within the grid, indicating the position of the current block within the grid. blockDim.x is the size of the block, indicating the number of threads in each block.

2.4. Parallel based on bonds

Since the function of bonds typically resides within the time-step

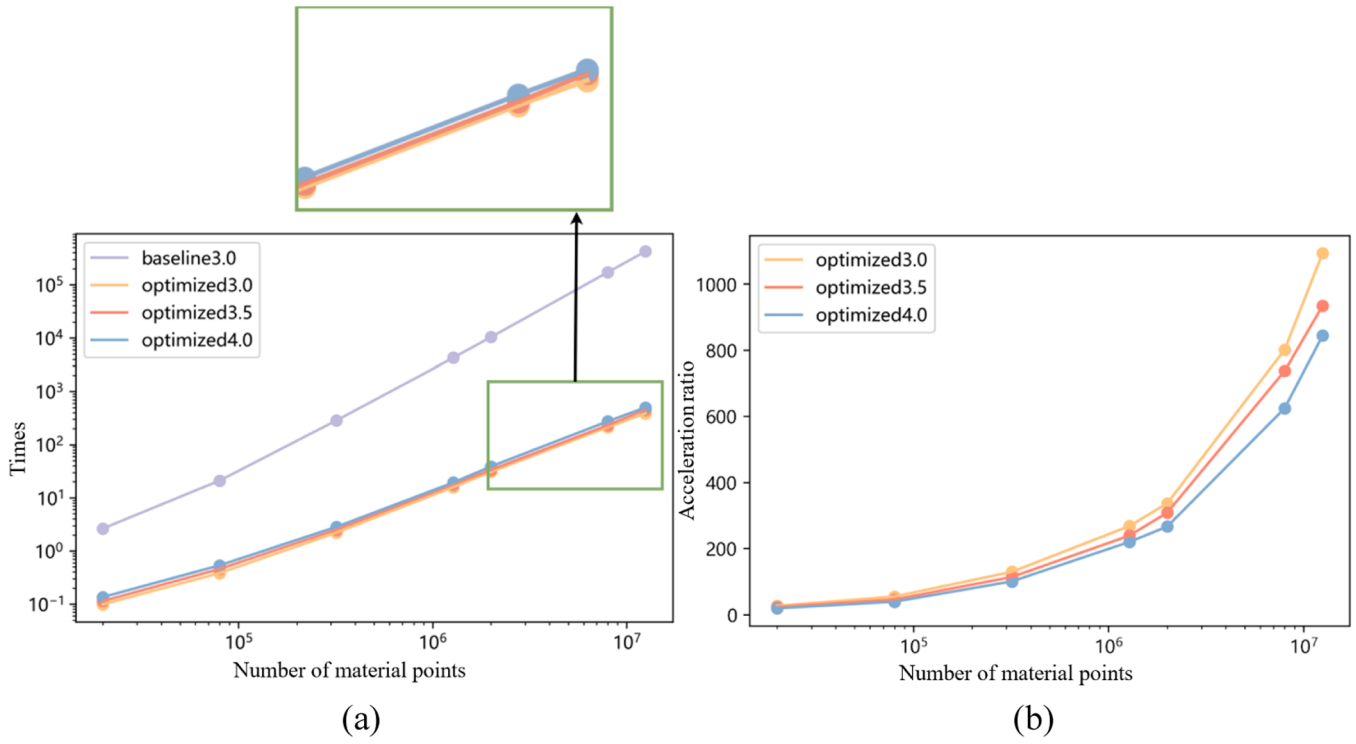

Fig. 2. Comparison between the parallel and series program of neighborhood generation module (a) Time consumption analysis of neighborhood generation(*ms*), (b) Speedup analysis of neighborhood generation module.

loop and is executed repeatedly, optimizing them can significantly reduce the overall computation time.

The important step is to set global thread coordinates. This involves defining the computational grid in CUDA, which determines how threads are mapped onto the GPU. Each thread is responsible for computing the state of a particular bond or set of bonds, ensuring that the computation is distributed across the GPU's processing units.

$$\text{gpuIdx} = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x}. \tag{7}$$

Then, the index of the present points is changed as

$$\text{nodeIdx} = \text{gpuIdx}/\text{maxNeighbor}. \tag{8}$$

The neighborhood index of present point is:

$$\text{targetIdx} = \text{gpuIdx}\%\text{maxNeighbor}. \tag{9}$$

In the implementation, maxNeighbor is an estimated value for the number of neighbors in the model. This transforms the original thread count into $P \times$ maxNeighbor, representing the estimated number of bonds, where P stands for the total number of material points. The pseudocode for parallelizing computations related to bonds is presented in Algorithm 2. During the computation, all bond data must be updated to their corresponding points. Since the concurrent execution of each thread can potentially affect the global particle information, the program uses atomicAdd for mutual exclusion when writing to global memory. This ensures that only one thread can modify shared data at any given time, thus preventing data races, as indicated in line 7.

2.5. Parallel of Neighborhood Generation Function

In order to parallelize the program, it is necessary to separate the neighborhood list. Since the exact number of domain points is not known in advance, it is necessary to allocate enough memory and launch threads based on the estimated number of neighborhood points. This results in some waste memory space and thread resources. As a solution, this process is broken down into three functions. First, obtain the number of neighborhoods for all points; then use the determined maximum number of neighborhood points to replace the original estimated value; Finally, generate neighborhood points based on the allocated memory, as shown in Algorithm 3.

Breaking down the neighborhood generation function into three parts, the corresponding time complexity becomes twice that of the original. This may result in more time spent on the neighborhood generation part. However, since this part of the function only needs to be executed once in the entire program, the reduction in memory and the savings in execution resources are an improvement for the subsequent loops that need to be performed thousands of times, so the overall efficiency is higher. Additionally, two optimization techniques are employed to accelerate this function.

The semi-neighborhood technique is only applicable to PD algorithms in which the neighborhood points remain unchanged during the computation. The neighborhood points are determined based on the distance of a bond. This distance is bound to the geometric structure and will not change during the iteration process. It is important to note that if point j is within the neighborhood space of point i, then point i must also be within the neighborhood space of point j. As a result, the original time complexity can be reduced from O(P^2) to O(P^2/2). The other is pruning method, when a point has already appeared in the neighborhood of a target point, and if there are consecutive (2*estimate-1) points that are not in the neighborhood of the target point (the estimated value is set to the number of horizontal points in the geometry), then jump directly to the next target point, and prune the subsequent points.

To test the efficiency of the optimized neighborhood generation function, the executive time and acceleration ratio of optimized parallel program with δ values of 3.0dx, 3.5dx and 4.0dx are compared to that of series program with a δ value of 3.0dx.

The results are depicted in Fig. 2, where "optimized" refers to the optimized algorithm, and "baseline" refers to the original program. A comparison of the two shows that the optimized algorithm has significantly improved in the neighborhood generation module. In Fig. 2 (a), as the number of points increases, the time consumption of the optimized

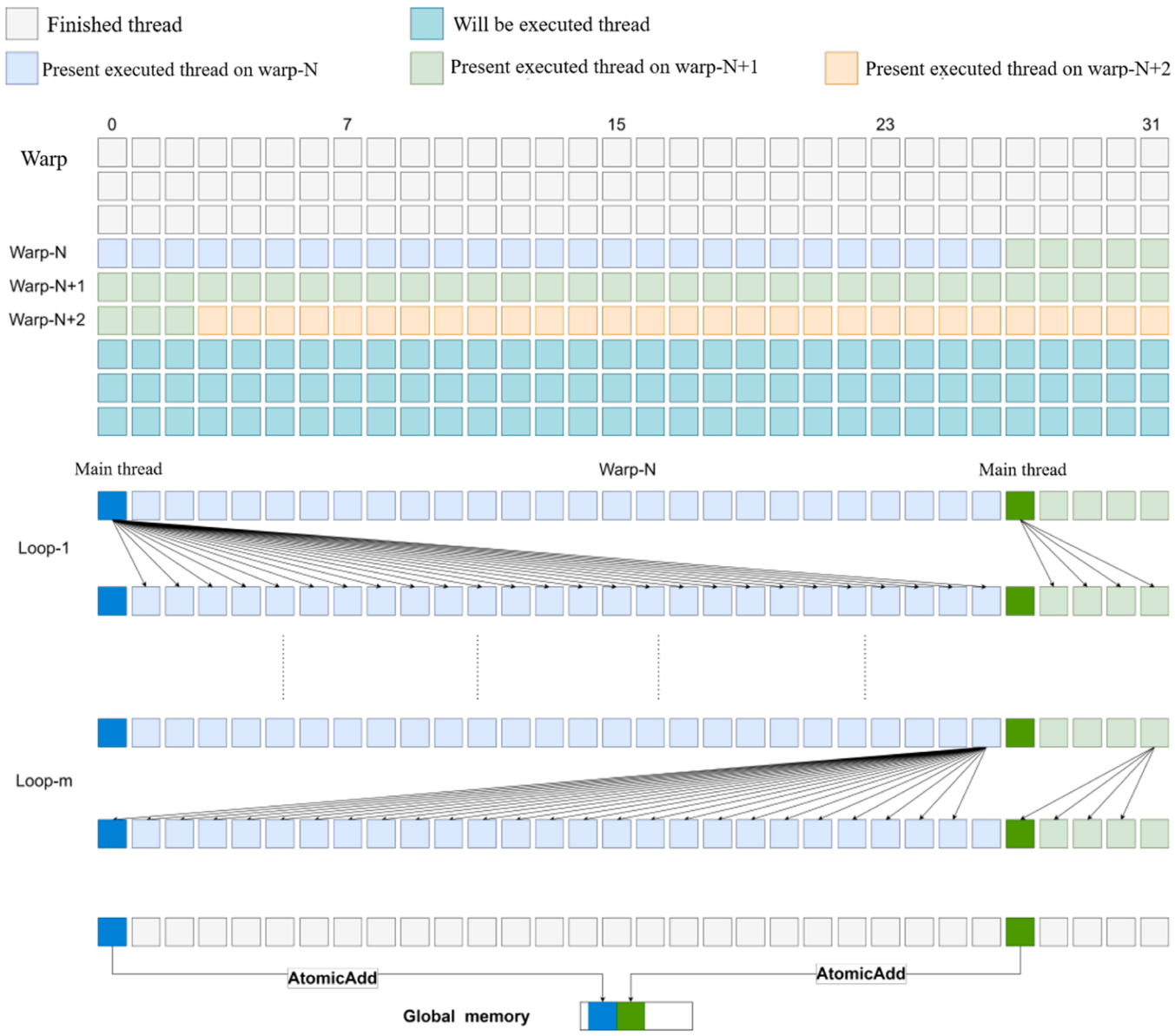

Fig. 3. The schematic of a general register memory access.

algorithm also increases, but at a slower rate compared to the baseline program. The time spent generation for neighborhoods gradually increases as the neighborhood radius expands. In Fig. 2 (b), when the number of points is below 10^5, there's a tenfold time difference. As the number of points increases to 10^6, the difference becomes more than 100 times, and after 10^7, it even achieves an acceleration efficiency of over 1000 times.

2.6. Optimization strategy of memory access

The parallel algorithms mentioned above all make use of global memory, which does not fully utilize memory resources and still has significant optimization potential. Registers in the CUDA memory structure have the largest memory bandwidth and the smallest memory latency. In the CUDA sm_89 architecture, a thread block can utilize up to 65,536 register resources. For calculating internal forces, each thread requires 32 registers, totaling 20,480 for a maximum complete thread block with 1024 threads. The number is well below the available register count, preventing register overflow. Therefore, utilizing the bandwidth of registers for high-speed memory access is a more efficient strategy. In CUDA, a warp is the smallest hardware unit of execution, and the register resources it occupies can exchange information with each other without needing to access global memory. Thus, this section utilizes warp functions to implement data exchange within the warp, reducing the need to access global memory by exchanging data in registers.

In the internal force calculation of a single particle, where a particle requires launching MaxNeighbor threads. First, classify the threads within the same warp and match them according to the index of the current point, as well as use a 32-bit unsigned number as a mask for storage. When the bit is set to 1, it represents that thread will perform an accumulation. For example, in the computation process of warp-N shown in Fig. 3: the blue part matches the first 27 threads, which calculate the bonds of the same point, using the binary expression mask-1 = 1111 1111 1111 1111 1111 1111 1110 0000. The green part matches the last 5 threads, which calculate the key information of the next point, using the binary expression mask-2 = 0000 0000 0000 0000 0000 0000 0001 1111. Then, select the leading thread from the warp. By default, the first channel that matches the same nodeIdx in the warp is chosen. For example, channel 0 in mask-1 is the leading thread for this computation, and channel 27 in mask-2 is the leading thread for the

Algorithm 4
A general register memory access optimization.

```
Data: gpuIdx, threadIdx.x, blockIdx.x, blockDim.x, maxNeighbor, nodeIdx, targetIdx,
      laneIdx, warpSize, res, nt, numfam[], mask, offset
Result: res
1  gpuIdx ← threadIdx.x + blockIdx.x * blockDim.x;
2  nodeIdx ← gpuIdx / maxNeighbor;
3  targetIdx ← gpuIdx % maxNeighbor;
4  laneIdx ← gpuIdx % warpSize;
5  res ← 0.0;
6  mask ← _match_all_sync(nodeIdx);
7  if nodeIdx < nt and targetIdx < numfam[nodeIdx] then
8      res ← Calculation( );
9      _syncwarps(); // (waiting until all threads in same warp executed here)
10     if mask == 0xFFFFFFFF then
11         while offset ≥ 1 do // operating register in the same warp to add
12             res+= shfl down sync(mask, res, offset, warpSie):
13             offset /= 2;
14         end
15     end
16     else
17         while offset ≥ 1 do // operating register in the same warp to add
18             res += _shfl_sync(mask, res, offset, warpSize);
19             offset >> 1 ;
20         end
21     end
22     if laneldx == getFirstMatchLane(mask) then
23         atomicAdd(&target[nodeIdx], res);
24     end
25 end
```

second part. Next, use the _shfl_sync() function to broadcast the data calculated by the channels marked as 1 in the mask to the leading thread and accumulates to obtain the final result. Since the computation of each point may cross threads, a plus operation is required to update the global data. The pseudocode is detailed in Algorithm 4-4, it is important to note that the thread matching function _match_all_sync() used in step 6 requires a CUDA compute capability greater than 7.0. Additionally, all threads in this method execute in parallel, which can lead to data racing issues when reading and writing. Writing to the same memory location at the same time can result in inconsistent data. Therefore, atomicadd is used when writing back to global memory. Atomicadd prevents the data at the memory address being added to from being overwritten.

Integrating the aforementioned parallel strategies and optimization techniques, a low-cost, high-efficiency BBPD algorithm parallelized on GPU has been designed and named PD-general.

3. Numerical analysis and discussions

The developed parallel program is implemented with a GeForce RTX 4070 GPU. This GPU features 46 streaming multiprocessors, 5888 CUDA cores, and uses CUDA version 12.2. It is based on the sm_89 architecture and has a maximum global memory size of 12GB, with a memory bandwidth of 469.43GiB/s, and a theoretical bandwidth is 502.73GiB/s. It should be noted that due to the use of a PCI-E 3.0 slot on the

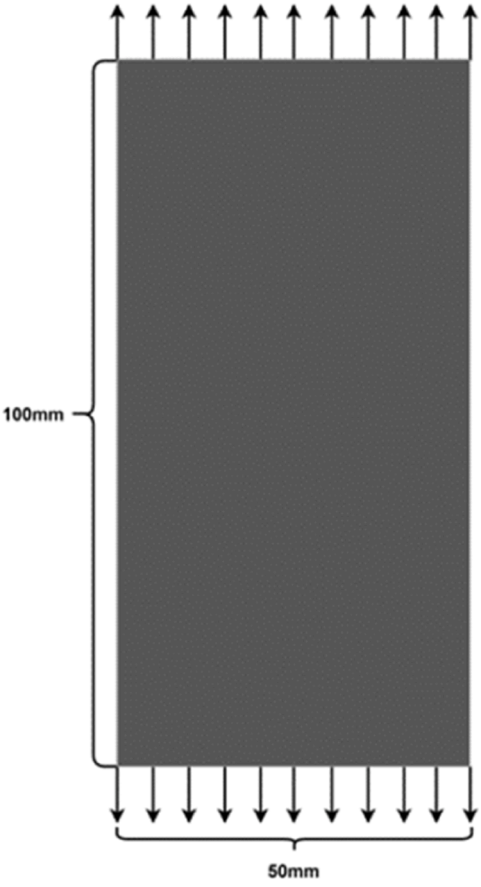

Fig. 4. The schematic of a two-dimensional plate with uniaxial tension.

motherboard, there is some bandwidth loss. The constant memory size of the graphics card device is 64KB. The CPU platform used is an Intel Core i7–10,700, an octa-core, sixteen-thread processor with a base frequency of 2.90 GHz. The memory size is 64GB with a memory bandwidth of 45.8GiB/s. The operating system executed is Ubuntu 22.04.3 LTS. After exploring the optimal thread organization for the program, a configuration of 256 threads was chosen. In subsequent studies, all CUDA functions will adopt the block configuration of (256, 1, 1). Two examples are calculated to verify the validation of parallel algorithms of the basic BBPD and BBPD with cracks.

3.1. Uniaxial tension of a two-dimensional plate

A two-dimensional plate with dimensions of 100mm × 50mm is depicted in Fig. 4. The plate has a mass density of $\rho = 5000$ kg/m^3, an elastic modulus of $E = 100$ GPa, and a fixed Poisson's ratio of 1/3. Opposite loads of magnitude σ =100 MPa are applied to the top and bottom sides. The neighborhood radius $\delta = 3.17dx$. The time step is $dt = 0.001\ s$, and the total number of iterations is 6000. The bond constant is $c = 315.0 \times E/(8.0 \times \pi \times \delta^3)$.

Three different sets of data with 5000, 125,000, and 1000,000 discrete points are being analyzed, respectively. The experimental results can be seen in Fig. 5, which shows that as the number of points increases, the results become smoother and more accurate.

Based on varying numbers of material points and neighborhood sizes, the error analyses are conducted. The time step is 4000, and for the example calculation, the neighborhood radii are set as 3.1dx and 3.2dx as shown in Fig. 6. It is evident that the error decreases as the number of points increases. Furthermore, with a larger neighborhood radius, the corresponding error decreases as well, due to the increased number of neighborhood points.

The displacements of top material points are used to test the convergence of the program with different time iterative steps. Five discretization models with 1250, 5000, 20,000, 80,000, and 320,000 points are being analyzed to compare with the theoretical solution. The number of iterations is set at 4000 steps. Fig. 7 shows that as the time step progresses, the results initially exhibit oscillations and then tend towards the theoretical solution. The results approach stability by the 2750th step. Additionally, as the number of discrete material points increases, the results more closely approximate the analytical solution.

3.2. Kalthoff-Winkler Crack Propagation

Fig. 8 shows the experimental setup for the numerical validation of Kalthoff-Winkler. The dimensions are10mm × 20mm. Two parallel cracks, each 5mm in length, were created at distances of 7.5mm and 12.5mm from the left boundary at the top. An initial downward velocity boundary condition of $v = -10\ m/s$, was applied in the middle of the two cracks. The material density is $\rho = 7800\ kg/m^3$, the elastic modulus is $E = 190\ GPa$, the time step is $dt = 0.01us$, and the total number of time steps is 1000.

The results are displayed in Fig. 9, indicating that cracking begins around the 190th time step and continues to propagate. By the time step 870, a rupture has occurred at an angle of approximately 110° to the pre-existing cracks. These results are consistent with the outcomes provided in the Kalthoff-Winkler experiment [34] in Fig. 10, confirming the accuracy of the parallel program of BBPD with cracks.

4. Discussions

4.1. Most effective thread performance

The study aims to find the most effective thread organization for the program. The smallest execution unit on the GPU is a warp, which consists of 32 threads. Therefore, the thread blocks are organized as 32k (k is a positive integer). The study compares the performance of different thread block organizations in three models, using the computation of internal forces as the baseline. The testing will be conducted using Nsight systems, analyzing the execution of a single time step. Thread block organizations of 64, 128, 256, 512, 768, and 1024 threads will be tested, and the results are shown in Fig. 11. The bar chart describes the occupancy rate, while the line chart indicates execution efficiency.

When the number of threads per block is 1024, it results in the lowest performance. The higher the occupancy rate of the thread block organization, the greater the computational efficiency. In general, the organization with 128 to 256 threads delivers the best performance. Taking into account the potential impact of an increased neighborhood radius, the organization with 256 threads is chosen. Subsequent studies will be based on this test, and all CUDA functions will use the form of *block*=(256, 1, 1).

4.2. Overall Run Time

In this section, the total computation time of the program will be examined. The overall computation time is defined as the complete execution time from the start to the end of the program, including all operations except for data output. The experiment involves running three models on three different execution platforms: linear programs, OpenMP parallel programs, and CUDA parallel programs, which are respectively named: Serial, OpenMP, CUDA. The experiment covers seven groups of discrete structures of different scales, with the number of points ranging from 20,000 to 2000,000.

The study analyzed cases where the number of neighborhood points was greater than 32 and less than 32. Fig. 12 (a) and (c) indicate that the CUDA-based parallel program is significantly more efficient than the OpenMP and series programs. Additionally, as the number of the material points increases, the efficiency of CUDA-based parallel program

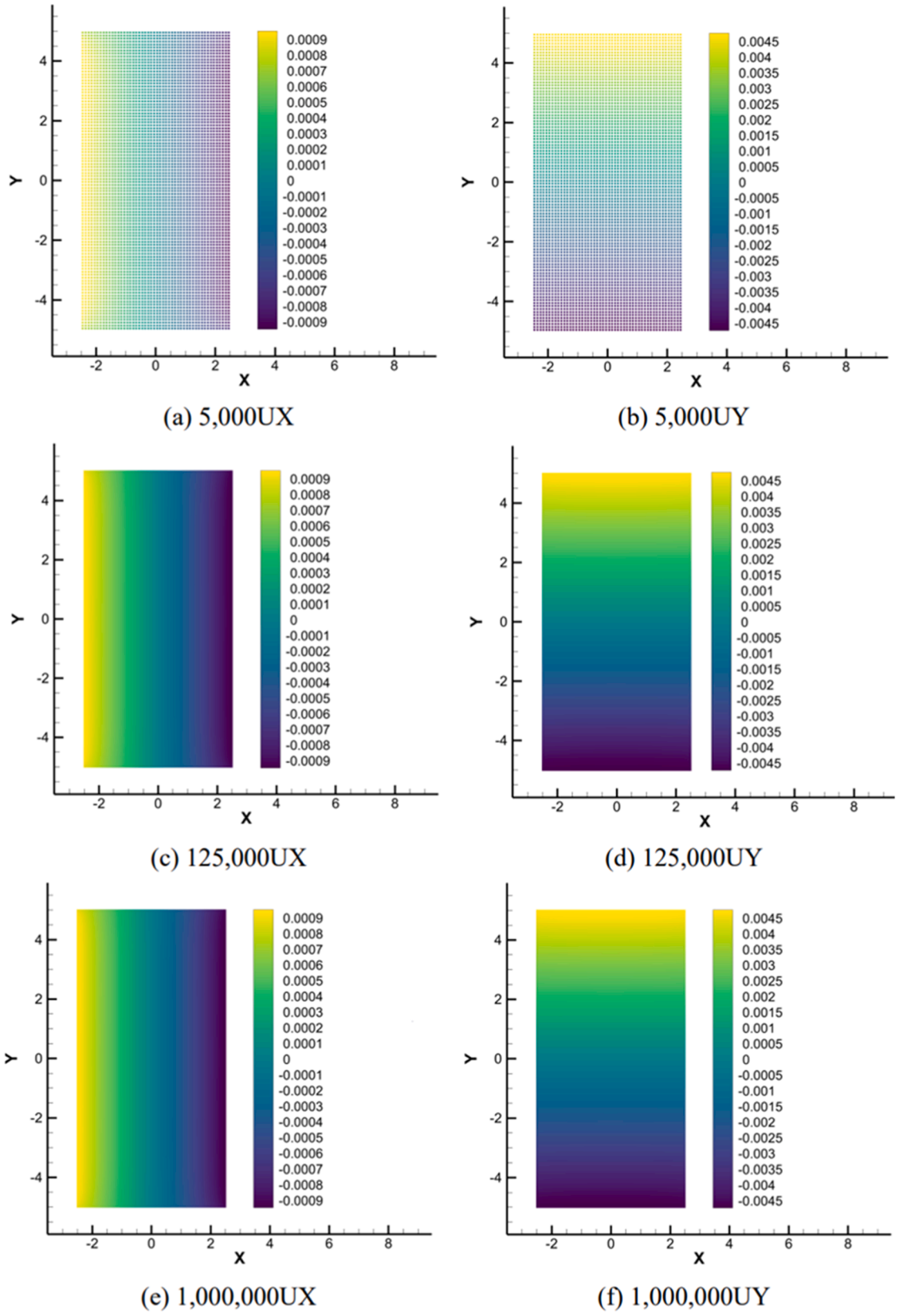

Fig. 5. Displacements of a two-dimensional plate with the uniaxial tension (a)(c)(e) horizontal displacements, (b)(d)(f) vertical displacements.

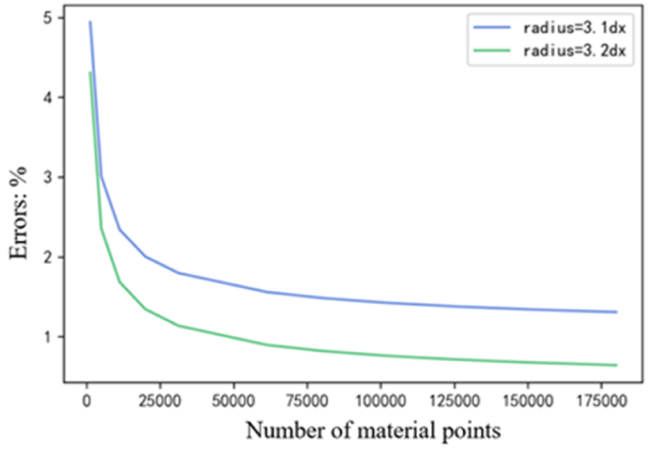

Fig. 6. Error analysis with different material points.

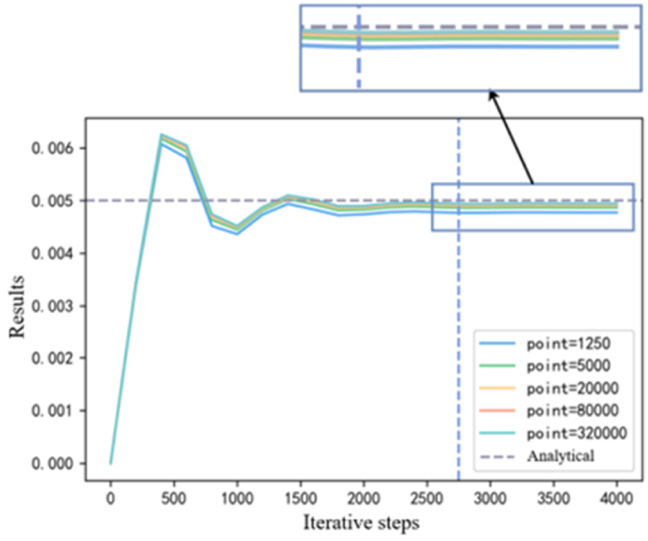

Fig. 7. Error analysis with different iterative steps.

becomes more noticeable. As depicted in Fig. 12(b) and (d), the initial acceleration ratio of the parallel program is relatively low, with no significant difference compared to the OpenMP program. However, as the material points increases, the acceleration ratio begins to rise. The acceleration ratio of BBPD with cracks can reach up to 200 times that of OpenMP.

The detailed execution time and acceleration ratio are listed in the Tables 2-3. The structure consists of 1000 × 2000 points with neighborhood radii of $\delta = 3.1$dx and $\delta = 3.2$dx, respectively. The time step is set at 0.001s, with a total of 1000 iterations.

Tables 2 and 3 show the execution times of the BBPD function modules, and all implementations have successfully achieved acceleration. Among them, the calculation of internal forces is the most time-consuming part, but it is also the part that accelerates the fastest, achieving a speedup of 314.61 times compared to the serial program.

Tables 4 and 5 show the results of the BBPD with cracks, this model demonstrates even faster acceleration compared to the previous one. However, as the number of horizontal points increases, the storage row becomes larger, making cache misses more likely. This is fatal for a CPU, but it has minimal impact on a GPU, as the pipeline execution can hide the latency of cache misses. The OpenMP-optimized parallel program is also affected by caching, but its acceleration ratio remains relatively stable.

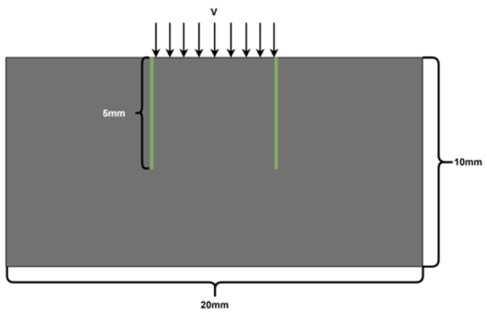

Fig. 8. Schematic of Kalthoff-Winkler.

4.3. Optimization Method Performance

In this section, the efficiency of the general register memory access module is tested. The scale of the selected case is 2000,000 points, the time step dt is 0.001, and the total number of iterations is 1000 steps. A comparison of all functions is made with the neighborhood exceeding 32 points. The functions were analyzed by linear programs, OpenMP programs, unoptimized GPU programs, and general register methods, which are named Serial, OpenMP, Bond, and General, respectively. The detailed results are shown in Fig. 13. The figure indicates that the optimized GPU strategy significantly reduces the computation time compared to the other programs.

Crack simulation, which skips computation when a failure occurs, leads to higher overall execution efficiency. This is especially true when all bonds fail. Despite appearing to have more memory access than BBPD, actual the execution speed is faster. However, this is may not be the case for CPUs, as they perform the exact same judgment every time. Even if they skip computational, the power may not be sufficient to cover the computation delay, resulting in low efficiency.

Fig. 14 shows the acceleration ratios for different optimization methods when the number of neighbors exceeds 32. The OpenMP parallel program consistently maintains an acceleration ratio of 5 to 9 times, while the General method can achieve a stable acceleration effect between 200 to 350 times. In contrast, the bond-based method performs weakly. These results highlight that the General method is effective in handling situations with a large number of neighbors.

4.4. Maximum Case Size Study

In this section, the maximum case size is investigated. It is known that the demand for global memory in the peridynamics framework is as follows:

$$\mathrm{BBPD} = 5\mathrm{ADP} + \mathrm{AP} + 4\mathrm{PN}, \tag{10}$$

$$\mathrm{BBPD/Crack} = 5\mathrm{ADP} + \mathrm{AP} + 5\mathrm{PN}, \tag{11}$$

where

D = dimension, one dimension=1, two dimension=2, three dimension=3;
P = Number of material points;
N = Number of neighborhood points
A = precision, single=4, double=8.

ADP, AP, and PN represent memory size measured in bytes. The GPU used in the experiment has a theoretical memory size of 12GB. However, due to the operating system's requirements and memory occupied by the CUDA launch, only about 11GB of memory is available for programming. The following calculations will use 11GB as the baseline to determine the maximum scale of the PD-General. By substituting the four conditions: 1. Double precision, N = 28; 2. Double precision, N =

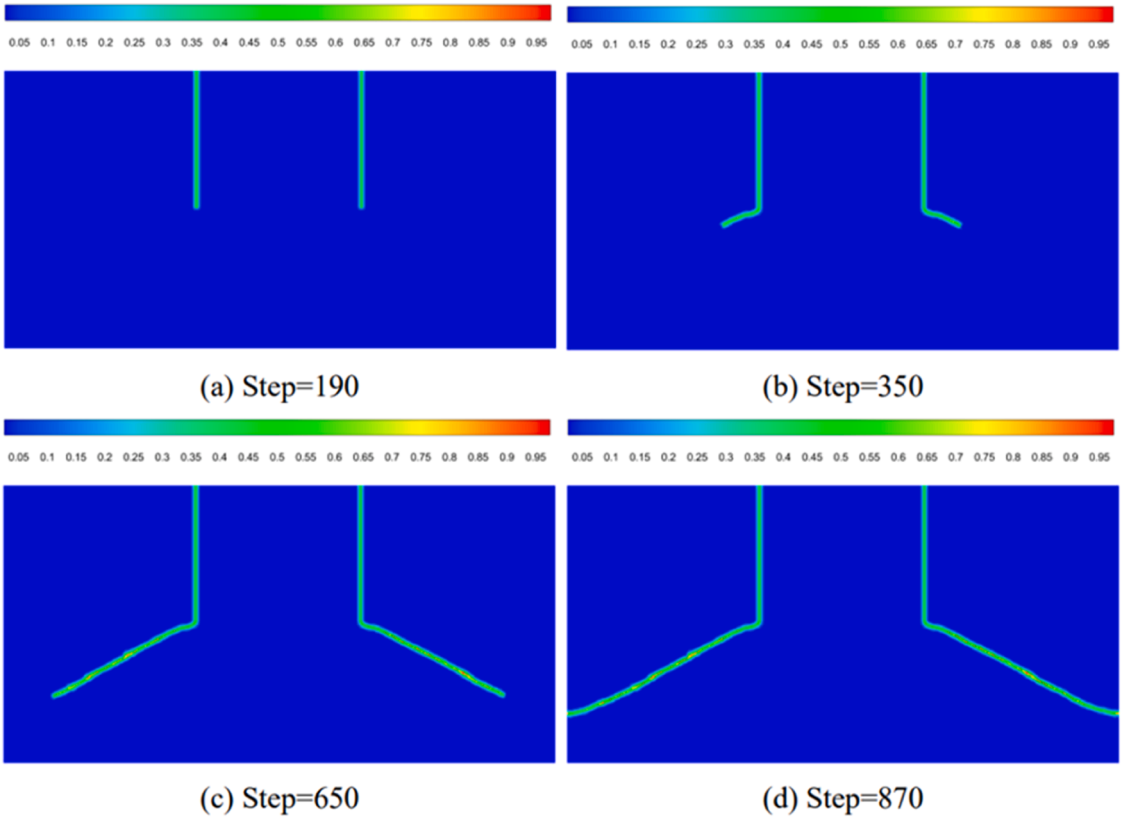

Fig. 9. Crack propagation processes.

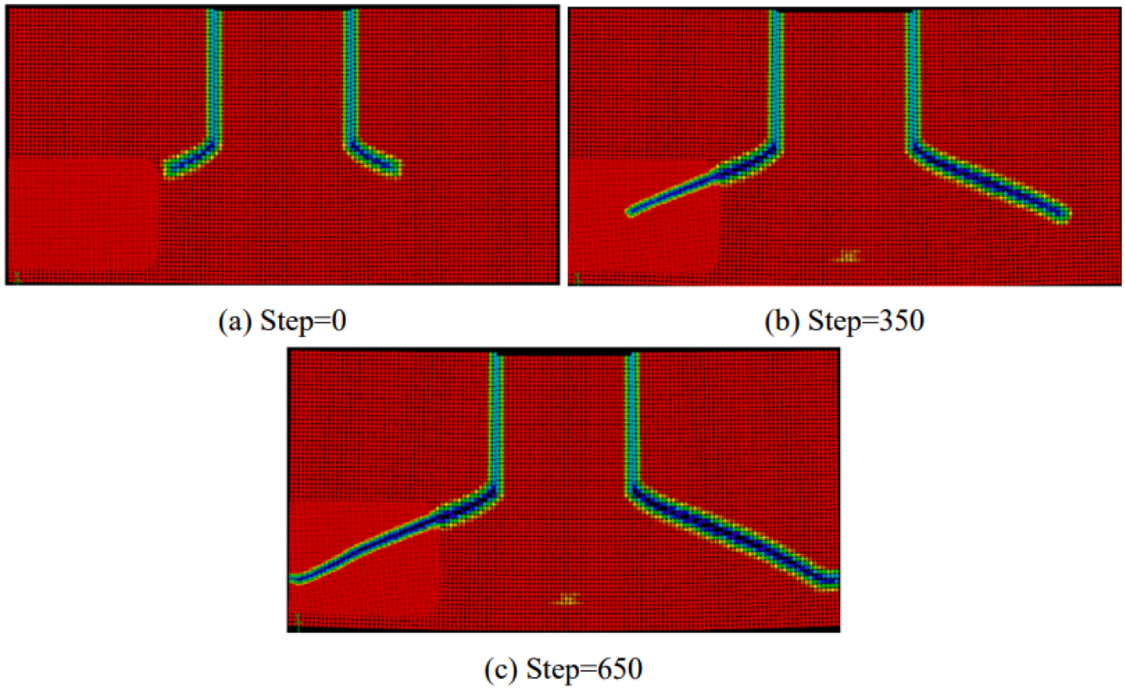

Fig. 10. Results of Kalthoff-Winkler experiment [34].

36; 3. Single precision, N = 28; 4. Single precision, N = 36 into the corresponding memory calculation formulas, the theoretical memory occupancy under the corresponding conditions is obtained as follows in Table 6.

Start with the theoretical results from the table mentioned earlier as the initial scale. Increase the number of points with each iteration until the program crashes. Test the two models mentioned above, and the calculation results are shown in the Table 7 below:

According to official Nvidia data, the GPU environment offers greater support for single precision, with single-precision FLOPS at

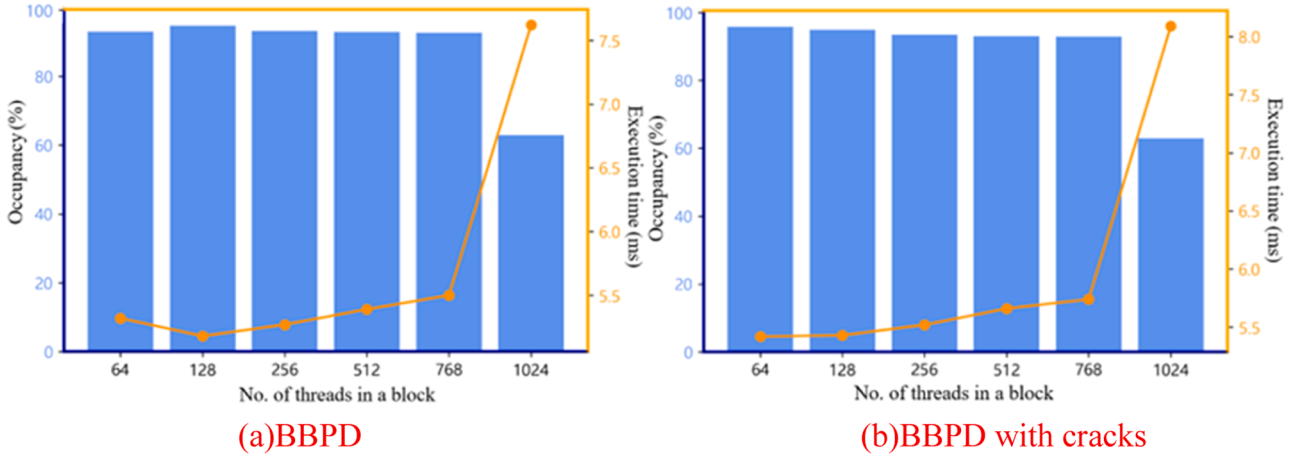

Fig. 11. Thread Organization and Program Performance Relationship.

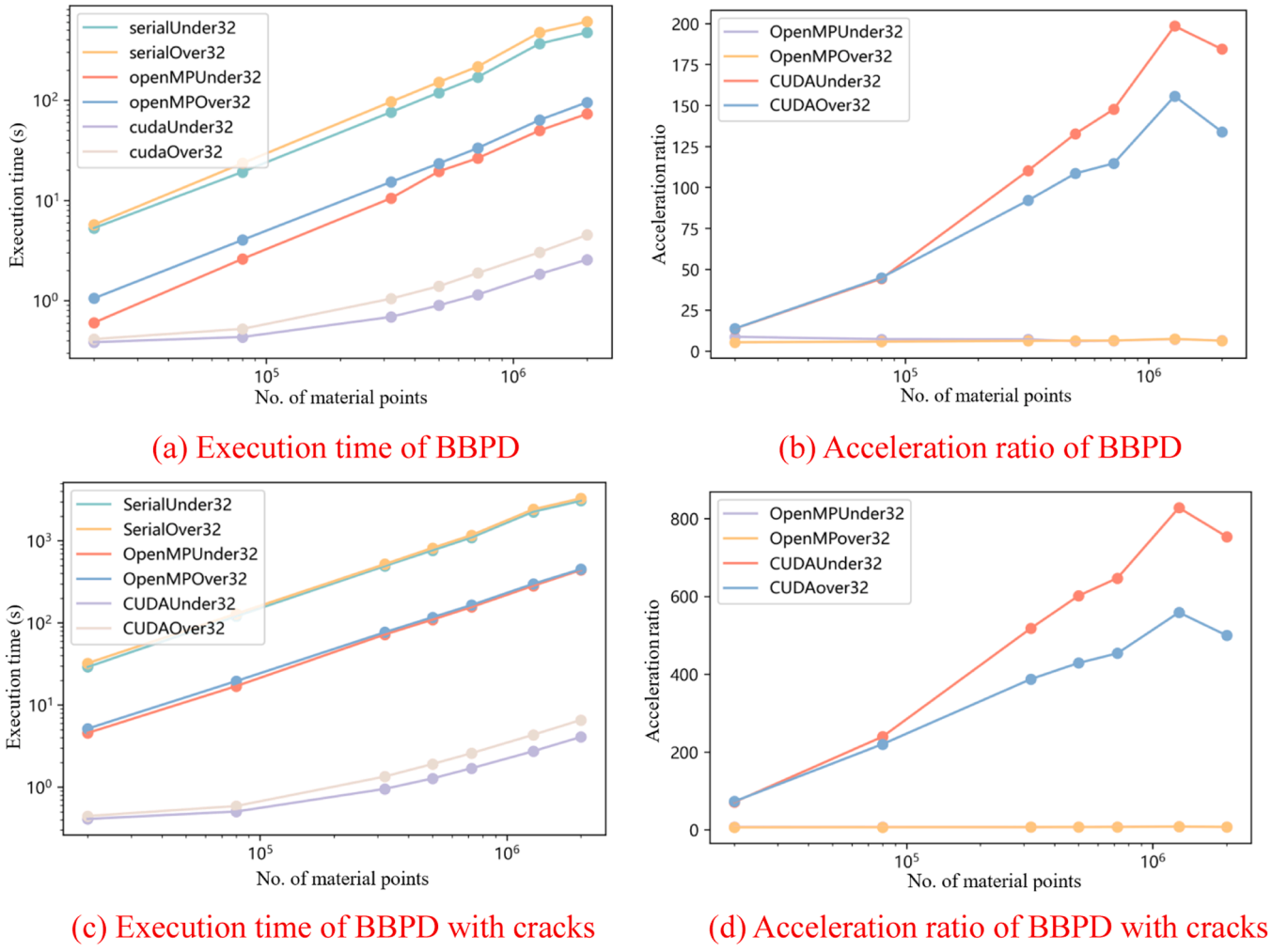

Fig. 12. Overall Execution Situation.

29.15 TFLOPS, while double-precision FLOPS are 455.4 GFLOPS. This means that the throughput of single precision is 64 times that of double precision. Consequently, at higher precision levels, double precision processing runs slower when handling the same scale problem. The size of the neighborhood radius also affects program performance. Tables 6-7 demonstrate that a greater number of neighborhood points results in a smaller execution scale and reduced execution efficiency.

For large-scale cases, the PD-General models described in this paper are capable of simulating tens of millions of particles. With optimized parameter settings, the maximum number of particles that can be simulated is 75,645,000. Specifically, in single precision, simulating 1000 steps can be completed in just 100 to 600 seconds, while in double precision, the execution time ranges from 900 to 2500 seconds. It is important to note that this time frame is generally kept within an hour.

Table 2
Execution time and acceleration ratio of BBPD with maxNeighbor = 28.

Functions	Execution time (*ms*)			Acceleration ratio	
	Serial	OpenMP	PD-General	OpenMP	PD-General
Discretization	1.84	1.59	1.58×10^{-1}	1.15	11.65
Neighborhood generation	1.19×10^{4}	1.94×10^{3}	4.32×10^{2}	6.16	27.61
Internal force	5.12×10^{5}	6.79×10^{4}	1.63×10^{3}	7.54	314.61
Update displacements	4.34×10^{3}	3.42×10^{3}	1.72×10^{2}	1.27	25.26

Table 3
Execution time and acceleration ratio of BBPD with maxNeighbor = 36.

Functions	Execution time (*ms*)			Acceleration ratio	
	Serial	OpenMP	PD-General	OpenMP	PD-General
Discretization	1.86	1.62	1.58×10^{-1}	1.14	11.78
Neighborhood generation	1.20×10^{4}	1.95×10^{3}	4.46×10^{2}	6.16	26.93
Internal force	6.59×10^{5}	8.24×10^{4}	3.74×10^{3}	8.00	176.11
Update displacements	4.33×10^{3}	3.09×10^{3}	1.71×10^{2}	1.40	25.28

Table 4
Execution time and acceleration ratio of BBPD with cracks with maxNeighbor = 28.

Functions	Execution time (*ms*)			Acceleration ratio	
	Serial	OpenMP	PD-General	OpenMP	PD-General
Neighborhood generation	3.38×10^{5}	5.44×10^{3}	1.14×10^{2}	6.21	297.87
Initial crack	8.96×10^{2}	1.32×10^{2}	1.78	6.81	503.89
Internal force	2.13×10^{6}	2.78×10^{5}	1.95×10^{3}	7.83	1091.13
Displacement update	3.92×10^{3}	2.93×10^{3}	1.29×10^{2}	1.34	30.26
Crack judgement	9.15×10^{5}	1.37×10^{5}	1.43×10^{3}	6.68	638.22
Damage judgement	1.25×10^{5}	2.14×10^{4}	1.45×10^{2}	5.84	863.34

Table 5
Execution time and acceleration ratio of BBPD with cracks with maxNeighbor = 36.

Functions	Execution time (*ms*)			Acceleration ratio	
	Serial	OpenMP	PD-General	OpenMP	PD-General
Neighborhood generation	3.38×10^{5}	5.43×10^{3}	1.14×10^{2}	6.22	296.90
Initial crack	9.52×10^{2}	1.40×10^{2}	2.28	6.79	417.09
Internal force	2.27×10^{6}	2.97×10^{5}	4.08×10^{3}	7.64	556.04
Displacement update	4.09×10^{3}	2.92×10^{3}	1.29×10^{2}	1.39	31.77
Crack judgement	9.76×10^{5}	1.45×10^{5}	1.83×10^{3}	6.75	532.44
Damage judgement	1.32×10^{5}	3.18×10^{4}	1.83×10^{2}	4.15	721.30

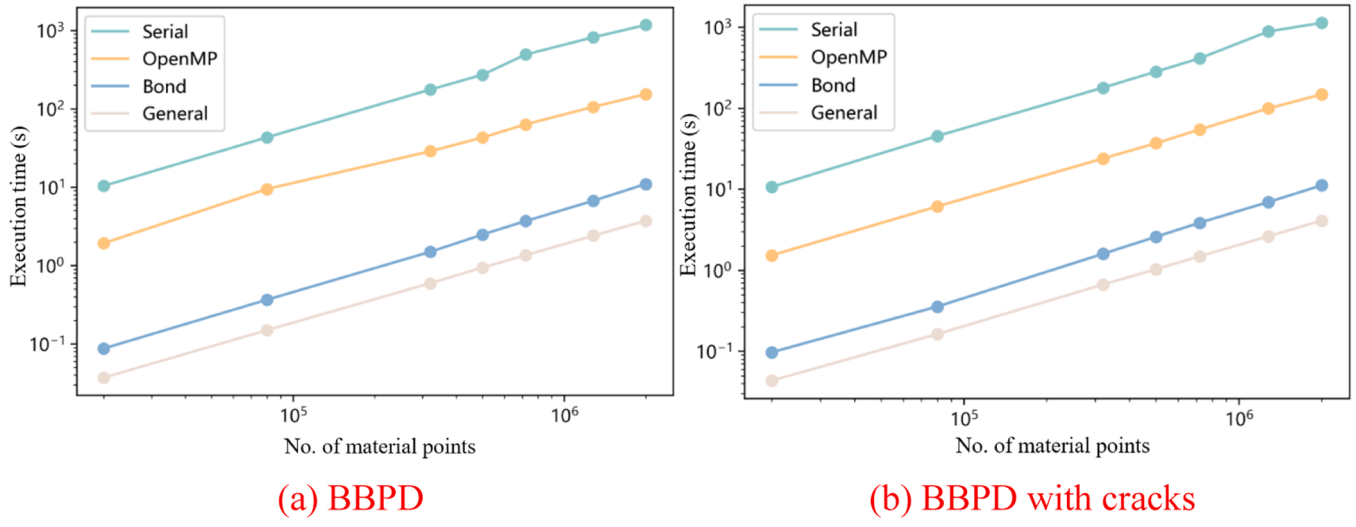

Fig. 13. Optimization Method Performance Comparison.

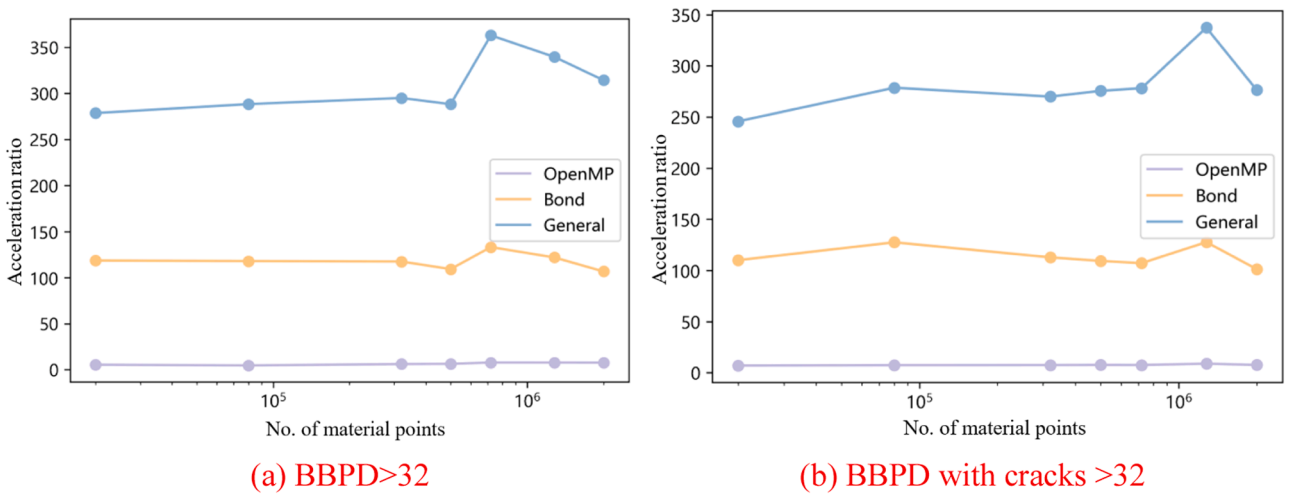

Fig. 14. Optimization Method Acceleration Ratio Comparison Chart.

Table 6
Precision and Neighborhood Size Comparison.

Precision	Neighborhood	BBPD	BBPD with crack
Double	δ=3.1dx	200P	228P
Double	δ=3.2dx	232P	268P
Single	δ=3.1dx	156P	184P
Single	δ=3.2dx	188P	224P

Table 7
Maximum Case Size Table.

Modes	No. of points	Precision	No. of Neighborhood	memory	PD
BBPD	50,904,050	double	36	10.98	1109.21
	58,969,800	double	28	10.99	663.54
	62,720,000	float	36	10.93	183.39
	75,645,000	float	28	10.9	120.35
BBPD with cracks	44,067,272	double	36	10.98	1346.26
	51,795,842	double	28	10.99	952.913
	52,647,848	float	36	10.97	176.145
	64,184,450	float	28	10.94	132.569

5. Conclusions

By analyzing the peridynamics model and the exploring parallel computing theory, a high-performance, low-cost peridynamics analysis framework called PD-General has been implemented using CUDA. It achieves two models of BBPD, and BBPD with cracks. In the parallel framework, the developed neighborhood generation module effectively reduces memory occupation waste. The general memory access module significantly improves the computational efficiency. This allows for expanded computational capabilities on personal computers. It reduces the execution time of CPU programs from several days to just a few hours or even tens of minutes using ordinary PC-level graphics cards, which is significant for advancing the PD research. Furthermore, PD-General can achieve acceleration effects of hundreds of times greater than serial programs and OpenMP parallel programs. Finally, the maximum device case exploration is conducted for these two models.

CRediT authorship contribution statement

Yang Yang: Writing – review & editing, Writing – original draft, Supervision, Funding acquisition. **Zixin Su:** Software, Data curation. **Yijun Liu:** Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank the financial support from National Natural Science Foundation of China (Project Nos. 12272160 and 12372198) and the Shenzhen National Science Foundation (Project No. 20231129213819001).

Data availability

Data will be made available on request.

References

[1] Silling S A. Reformulation of elasticity theory for discontinuities and long-range forces. J Mech Phys Solids 2000;48(1):175–209.
[2] Jafarzadeh S, Larios A, Bobaru F. Efficient solutions for nonlocal diffusion problemsvia boundary-adapted spectral methods. J Peridynam Nonlocal Model 2020;2:85–110.
[3] Jafarzadeh S, Wang L, Larios A, et al. A fast convolution-based method for peridynamic transient diffusion in arbitrary domains. Comput Methods Appl Mech Eng 2021;375:113633.
[4] Jafarzadeh S, Mousavi F, Larios A, et al. A general and fast convolution-based method for peridynamics: applications to elasticity and brittle fracture. Comput Methods Appl Mech Eng 2022;392:114666.
[5] Jafarzadeh S, Mousavi F, Wang L, et al. PeriFast/Dynamics: a MATLAB code for explicit fast convolution-based peridynamic analysis of deformation and fracture. J Peridynam Nonlocal Model 2023:1–29.
[6] Wang L, Jafarzadeh S, Mousavi F, et al. Perifast/corrosion: a 3d pseudo spectral peridynamic matlab code for corrosion. J Peridynam Nonlocal Model 2023:1–25.
[7] T Ni, M Zaccariotto, Q Zhu, et al. Matrix-based implementation and GPU acceleration of linearized ordinary state-based peridynamic models in MATLAB. 2023, arXiv: 2309. 11273.
[8] Liu W, Hong J W. A coupling approach of discretized peridynamics with finite element method. Comput Methods Appl Mech Eng 2012;245:163–75.
[9] Madenci E, Oterkus E, et al. Coupling of the peridynamic theory and finite element method. Springer; 2014. p. 191–202.
[10] Bobaru F, Yang M, Alves L F, et al. Convergence, adaptive refinement, and scaling in 1D peridynamics. Int J Numer Meth Eng 2009;77(6):852–77.
[11] Dipasquale D, Zaccariotto M, Galvanetto U. Crack propagation with adaptive grid refinement in 2D peridynamics. Int J Fract 2014;190:1–22.
[12] Galvanetto U, Mudric T, Shojaei A, et al. An effective way to couple FEM meshes and peridynamics grids for the solution of static equilibrium problems. Mech Res Commun 2016;76:41–7.
[13] Lee J, Oh S E, Hong J W. Parallel programming of a peridynamics code coupled with finite element method. Int J Fract 2017;203:99–114.

[14] Shojaei A, Zaccariotto M, Galvanetto U. Coupling of 2D discretized peridynamics with a meshless method based on classical elasticity using switching of nodal behaviour. Eng Comput 2017;34(5):1334–66.
[15] Fan H, Li S. Parallel peridynamics–SPH simulation of explosion induced soil fragmentation by using OpenMP. Comput Part Mech 2017;4(2):199–211.
[16] Bobaru F, Ha Y D. Adaptive refinement and multiscale modeling in 2D peridynamics. Int J Multiscale Comput Eng 2011;9(6):635–60.
[17] Yang Y, Liu YJ. Modeling of cracks in two-dimensional elastic bodies by coupling the boundary element method with peridynamics. Int J Solids Struct 2021;217-218:74–89.
[18] Yang Y, Liu YJ. Analysis of dynamic crack propagation in two-dimensional elastic bodies by coupling the boundary element method and the bond-based peridynamics. Comput Methods Appl Mech Eng 2022;399:115339.
[19] Nguyen C T, Oterkus S, Oterkus E. A peridynamic-based machine learning model for one-dimensional and two-dimensional structures. Continuum Mech Thermodyn 2023;35(3):741–73.
[20] Haghighat E, Bekar A C, Madenci E, et al. A nonlocal physics-informed deep learning framework using the peridynamic differential operator. Comput Methods Appl Mech Eng 2021;385:114012.
[21] Chen Y, Yang Y, Liu Y J. A neural network peridynamic method for modeling rubber-like materials. Int J Mech Sci 2024;273:109234.
[22] Y Yang, Y Chen, Y J Liu. A novel neural-network non-ordinary state-based peridynamic method for large deformation and fracture analysis of hyperelastic membrane. Comput Methods Appl Mech Eng 431, 117239.
[23] Wen-Mei W H, Kirk D B, EL Hajj I. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann; 2022.
[24] Mao Z R, Li X Y, Hu S Y, et al. A GPU accelerated mixed-precision smoothed Particle Hydrodynamics framework with cell-based relative coordinates. Eng Anal Bound Elem 2024;161:113–25.
[25] Littlewood D J. Roadmap for peridynamic software implementation. Albuquerque, NM (United States): Sandia National Lab. (SNL-NM); 2015.
[26] Sun L. Stability analysis method study of metalic plates based on peridyanmics. Shanghai Jiao Tong University; 2014.
[27] Liu S, Hu Y, Yu Y. Parallel computing method of peridyanmic models based on GPU. J Shanghai Jiaotong Univ 2016;50(9):1362–7.
[28] Li X, Ye H, Zhang J. Redesigning peridigm on SIMT accelerators for high-performance peridynamics simulations. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE; 2021. p. 433–43.
[29] Zhong J, Han F, Zhang L. Accelerated peridynamic computation on GPU for quasi-static fracture simulations. J Peridynam Nonlocal Model 2023:1–24.
[30] Mossaiby F, Shojaei A, Zaccariotto M, et al. Opencl implementation of a high performance 3d peridynamic model on graphics accelerators. Comput Math Appl 2017;74(8):1856–70.
[31] Boys B, Dodwell T J, Hobbs M, et al. PeriPy-A high performance OpenCL peridynamics package. Comput Methods Appl Mech Eng 2021;386:114085.
[32] Bartlett J, Storti D. A novel memory-optimized approach for large-scale peridynamics on the GPU. J Peridynam Nonlocal Model 2023;5(4):472–90.
[33] Wang X, Wang Q, Aa B, et al. A GPU parallel scheme for accelerating 2D and 3D peridynamics models. Theor Appl Fract Mech 2022;121:103458.
[34] Ren H, Zhuang X, Cai Y, et al. Dual-horizon peridynamics. Int J Numer Meth Eng 2016;108(12):1451–76.