Advances in Engineering Software 86 (2015) 29–38

Contents lists available at ScienceDirect

Advances in Engineering Software

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

A high performance crashworthiness simulation system based on GPU

Yong Cai [a], Guoping Wang [a,*], Guangyao Li [b,*], Hu Wang [b]

[a] School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
[b] State Key Laboratory of Advanced Design and Manufacturing for Vehicle Body, Hunan University, Changsha 410082, China

ARTICLE INFO

Article history:
Received 17 November 2014
Received in revised form 24 March 2015
Accepted 6 April 2015
Available online 22 April 2015

Keywords:
Crashworthiness
Explicit finite element
Graphics processing units
CUDA
Parallel programming
CAE

ABSTRACT

Crashworthiness simulation system is one of the key computer-aided engineering (CAE) tools for the automobile industry and implies two potential conflicting requirements: accuracy and efficiency. A parallel crashworthiness simulation system based on graphics processing unit (GPU) architecture and the explicit finite element (FE) method is developed in this work. Implementation details with compute unified device architecture (CUDA) are considered. The entire parallel simulation system involves a parallel hierarchy-territory contact-searching algorithm (HITA) and a parallel penalty contact force calculation algorithm. Three basic GPU-based parallel strategies are suggested to meet the natural parallelism of the explicit FE algorithm. Two free GPU-based numerical calculation libraries, cuBLAS and Thrust, are introduced to decrease the difficulty of programming. Furthermore, a mixed array and a thread map to element strategy are proposed to improve the performance of the test pairs searching. The outer loop of the nested loop through the mixed array is unrolled to realize parallel searching. An efficient storage strategy based on data sorting is presented to realize data transfer between different hierarchies with coalesced access during the contact pairs searching. A thread map to element pattern is implemented to calculate the penetrations and the penetration forces; a double float atomic operation is used to scatter contact forces. The simulation results of the three different models based on the Intel Core i7-930 and the NVIDIA GeForce GTX 580 demonstrate the precision and efficiency of this developed parallel crashworthiness simulation system.

1. Introduction

Crashworthiness simulation system is one of the key computer-aided engineering (CAE) tools for the automobile industry [1]. It is widely used in crashworthiness design [2–4]. With the usage of simulation system, an enormous amount of expensive and time-consuming physical tests are greatly reduced. However, due to the high nonlinearity of contact–impact problems, such simulations contain several computationally intensive parts such as Gaussian integration and contact searching. For example, a crashworthiness simulation of an energy-absorbing structure with a fine-meshed FE model usually consumes dozens of hours. Therefore, computational expense is a major bottleneck of the crashworthiness simulation in real engineering problems. The purpose of this work is to improve the efficiency of crashworthiness simulation under high accurate solution significantly.

Previously, the most of contributions for improving the performance of solver are based on the FE theories and the contact algorithms. Hughes [5–7] and Belytschko [8–10] proposed several advanced shell theories to improve the accuracy and stability of the nonlinear explicit FE method. A large number of researchers suggested several contact search algorithms for contact detection [11,12]. Zhong [13] published the first book on Finite Element Method (FEM) modeling of contact–impact events and deals with FE procedures for solutions to both static and dynamic contact–impact problems. On the other hand, supercomputing involving parallel computing has become research hotspots. Parallel computing is a direct way to improve the computational efficiency. The parallel implementations of shell element formulation have been reported in both literatures and commercial codes. The results of these studies show that the nonlinear explicit FE analysis can be accelerated by parallel computation significantly [14–16]. Therefore, many researchers make their efforts to realize the parallel implementation of contact searching. In the early stages, Belytschko et al. [17] presented a parallel explicit FE method for the contact–impact problem on a SIMD computer. Namburu and Turner [18] presented a contact–impact algorithm on a data-parallel computer. Besides, great efforts have been made to develop decomposed impact simulation algorithms executing on network-based parallel computing architecture, such as the multi-processors based architectures using distributed memory

* Corresponding authors.
E-mail addresses: wgp@pku.edu.cn (G. Wang), gyli@hnu.edu.cn (G. Li).

http://dx.doi.org/10.1016/j.advengsoft.2015.04.003
0965-9978/

processing (DMP) and the multi-cores based architectures using shared memory processing (SMP) [19–21]. These efficient parallel implementations enlarged the scale of CAE model, thus more complex structures could be modeled with fine meshes [21]. However, the disadvantages of these traditional parallel computations are obvious. Firstly, the hardware cost of CPU-based parallel architecture is too expensive, and it is more difficult to use special programming language to code parallel program based on supercomputers with hundreds or thousands of processors. Secondly, the administration and maintenance costs are super-linear rising along with growing demand for computation power. Therefore, an alternative way for CPU-based parallel program is needed urgently to reduce the cost of parallel computing. In the past couple of years, we turned our attention to general purpose computation on graphics hardware (GPGPU).

Nowadays, GPU offers a tremendous amount of computing resources not only for graphics processes but also for general-purpose parallel computations. These general parallel computing resources include massive processing cores, high memory bandwidth, and general-purpose instruction sets. In the GPU-based parallel program, GPU is commonly used as a coprocessor to execute easy parallel sections. For now, a large number of high performance implementations of FE applications based on GPU have been reported. Göddeke et al. [22] have successfully implemented their FE algorithm on a GPU enhanced cluster to solve implicit FE problem with multi-grid algorithm. For nonlinear FE analysis, Mafi and Sirouspour [23] proposed a GPU-based implementation of FE method using implicit time integration for dynamic nonlinear deformation analysis. Ikushima and Shibahara [24] presented an idealized explicit FEM accelerated by GPU to predicted the residual stresses in multi-pass welded joint. Furthermore, FE analysis involves fluid–solid coupling [25], structural analysis [26], higher order numerical integration [27] and others are parallelized by GPU successfully. Our research team also developed a GPU-based parallel sheet metal forming simulation system, which achieved up to 27X speedup using GTX285 GPU [28].

In this paper, GPU was used to implement the explicit FE method with a hierarchy-territory contact-searching algorithm (HITA) and a penalty contact force calculation algorithm to simulate car crash. The remainder of this paper is organized as follows. In Section 2, the mathematical model and numerical scheme of contact–impact problems are briefly introduced. In Section 3, the typical CUDA programming model and the details of the GPU implementation for the whole simulation algorithm are presented. Numerical experiments are used to evaluate the performance of the developed parallel simulation system in Section 4. Finally, conclusion remarks are presented in Section 5.

2. Mathematical model and numerical scheme

To simulate contact–impact process with FE method, contact boundaries are usually approximated by a collection of segments, and contacts are considered at the discrete contacting nodes. The most widely used segments of shell structure are triangular shell element and quadrilateral shell element, as shown in Fig. 1. The discretized contact solution divides the solution domain into discrete elements, and then, numerical interpolations are performed within these elements through shape functions. Furthermore, to facilitate the contact searching and contact force calculation, these segments usually correspond to a low order shell element, such as the Belytschko–Tsay (BT) element [8].

For convenience, contact segments are usually described as a multiple hierarchy system consists of segments, edges and nodes. When two body boundaries come into contact, one is specified as a master one, and the other is specified as a slave one. The segment, edge and node on master body called master segment, master contact edge and master contact node, respectively. Similarly, the segment, edge and node on slave body called slave segment, slave contact edge and slave contact node, respectively.

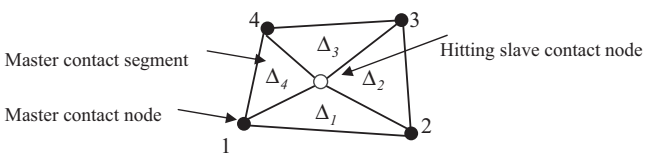


Fig. 1. A quadrilateral segment.

2.1. Contact searching algorithm

Master–slave algorithm is the most common algorithm for contact searching, which was first introduced by Hallquist [29]. However, this algorithm must specify master and slave segments respectively, and in some situations two boundaries may come into contact before the searching begins. These two drawbacks limit its ability to search contacts during the crashworthiness simulation, which usually has large displacements and rotations. Therefore, a more efficient and effective contact searching algorithm named HITA is used in this work [13].

The HITA method reduces the redundant contact searches by taking the advantage of a multiple-hierarchy contact system to improve its searching efficiency. In a hierarchical contact system, contact searching should be first performed between the higher level hierarchies and then between the lower level ones. If the contacts are rejected between two higher level contact segments, searching in the lower level hierarchies will not be performed. More importantly, the data independences of each segment in each hierarchy are well suited for GPU implementation. The following section provides a brief introduction of searching strategy based on the hierarchy territory techniques.

Firstly, the hierarchy territory of a segment is the smallest rectangular box which has its edges parallel to the global coordinate axes and contains the complete segment, as the solid line box shown in Fig. 2. Mathematically, it is a domain which can be defined as

$$T = \{(x_1, x_2) | x_i^a \le x_i \le x_i^b, i = 1, 2\} \tag{1}$$

where x_i^a and x_i^b denote the minimum and the maximum coordinates of node 1 and node 2, respectively.

Secondly, a contact territory is defined due to the rounding error of computer, as the shaded parts shown in Fig. 2. C_d is the control distance, and, C_p is the allowed maximum penetration.

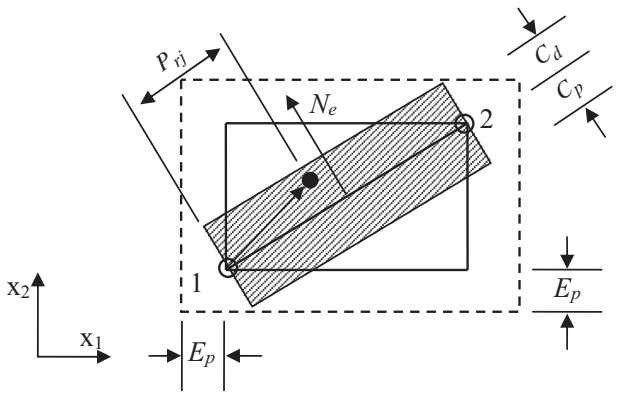


Fig. 2. Definition of different kind of territories for a segment.

With these two kinds of territories, a node lies within a segment's hierarchy territory to form a test pair, and, a node lies within a segment's contact territory to form a contact pair. The searching procedure is designed to consist of two main steps. The first step is to find all test pairs by calculating the intersection of hierarchy territories. The second step is to find all the contact pairs from the test pairs by calculating the exact position of node and its relevant target segment's contact territory. According to Fig. 2, it can be found that the hierarchy territory does not completely include the contact territory, so the hierarchy territories should be expanded sufficiently to form an expanded territory, which is denoted by

$$T_e = \{(x_1, x_2) | x_i^a - E_p \le x_i \le x_i^b + E_p, i = 1, 2\} \tag{2}$$

where x_i^a and x_i^b are indicated in Eq. (1), E_p denotes the amount of expansion, which is usually defined by the size of contact territory

$$E_p \ge \max(C_d, C_p) \tag{3}$$

Contact territories for three dimensional contact segments can be defined in a similar way. Consider a 3-node contact segment as shown in Fig. 3. The hierarchy territory is the smallest hexahedron box which has its faces parallel to the global coordinate planes and contains the complete segment, with a mathematical description as follows

$$T = \{(x_1, x_2, x_3) | x_i^{\min} \le x_i \le x_i^{\max}, i = 1, 2, 3\} \tag{4}$$

where $x_i^{\min}$ and $x_i^{\max}$ denote the maximum and the minimum coordinates of a segment, respectively. Correspondingly, the expanded territory can be expanded to

$$T_e = \{(x_1, x_2, x_3) | x_i^{\min} - E_p \le x_i \le x_i^{\max} + E_p, i = 1, 2, 3\} \tag{5}$$

The way to define a contact territory for a three dimension problem is a little more complex than two dimension problem, as show in Fig. 3(b). The mathematical description is

$$T_c = \{x | -C_p \le d(x) \le C_d, P_i(x) \le 0, (i = 1, 2, 3)\} \tag{6}$$

where

$$d(x) = (x - x_1) \cdot N_1 \tag{7}$$

$$P_i(x) = (x - x^i) \cdot \tilde{N}_g^i, (i = 1, 2, 3) \tag{8}$$

where x denotes the slave contact node coordinate, $\boldsymbol{N}_1$ is the unit normal vector of the master segment, $\boldsymbol{x}^i$ is the node coordinate of the master segment, and $\tilde{\boldsymbol{N}}_g^i$ is the unit normal vector of the contact edge.

2.2. The mathematical model of contact force

In the explicit FE method procedure, an efficient way to calculate the contact force is to allow the penetration between contacting boundaries and calculate the contact forces by

$$f_1 = -k_i p_i \tag{9}$$

where p_i is the penetration of contact segments and k_i is the contact stiffness of the contact segment. A typical contact stiffness of the shell contact segments is

$$k_i = \frac{f_s \times A_i \times K}{\min(\text{diagonal})} \tag{10}$$

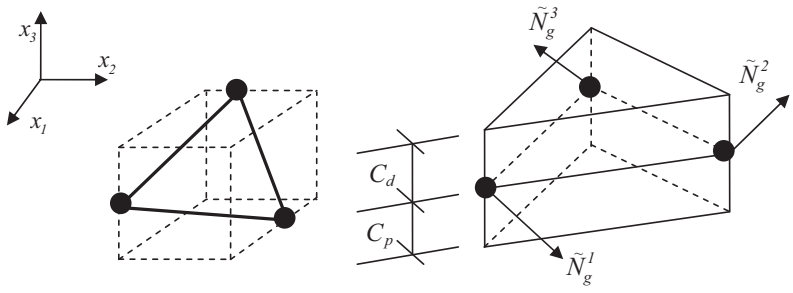
where f_s is the penalty scale factor, 0.1 by default. A_i is the area of the ith contact segment, K is the bulk module of contact segment, which is related to Young's module E and Poisson's ratio v. This method is known as penalty method, the contact force can be expressed by the displacement function and no new unknowns are introduced.

2.3. Distribution of the computing time

The main procedure of an explicit crashworthiness simulation program with above mathematical models is shown in Fig. 4. Firstly, a pre-searching is implemented to generate the contact hierarchies and other relevant information based on the FE mesh. Sequentially, the nodal external force and the internal force can be obtained. Then, the application begins to search contact pairs and compute the contact forces. The contact forces should be assembled into a global nodal force vector, and the physical quantities such as nodal velocities and coordinates must be updated based on the nodal force vector. Finally, the time-step size should be updated to guarantee the stability as the contact body becomes more and more distorted.

In general, the efficiency of a pure FE analysis program mainly depends on the size of elements. However, the contact algorithm is the major consuming part for a contact–impact problem. For example, as shown in Table 1, the contact searching consumed the most CPU time for an automobile body-in-white (BIW) crash model with 69,625 nodes and 65,177 elements. The cost for element formulations is the second-consumed one. Obviously, these two parts should be parallelized by GPU to improve the overall efficiency.

In addition, the data transfer between the host memory and the device memory are costly via a PCI Express whose peak theoretical bandwidth is only 16 GB/s or less. Hence, for a best overall application performance, it is important to minimize the data transfer between the host and the device. Therefore, although there is very little time consumed on computing the contact forces and other



(a) Hierarchy territory (b) Contact territory

Fig. 3. Contact territories for three dimension contact segment.

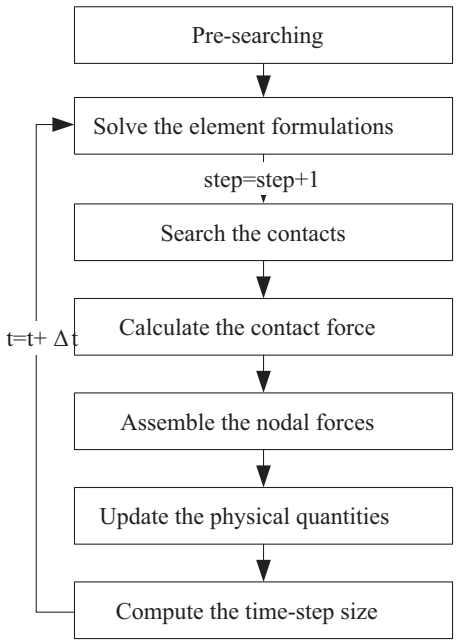


Fig. 4. Main procedures of a crashworthiness simulation.

Table 1
Computing time distribution for a BIW crash model.

Module	Clock cycles	Percentage (%)
Element formulations	140	21.88
Contacts searching	478	73.51
Contact forces	14	2.19
Others	8	1.25

small computing procedures, it is also important to parallelize them.

3. GPU implementation

3.1. GPU programming with CUDA

GPUs are parallel devices of the single instruction, multiple data (SIMD) classification, which are well suited for the problem that can be expressed as data-parallel computations with a high arithmetic intensity. CUDA programming involves running code on two different platforms concurrently: a host system with one or more CPUs and one or more CUDA-enable GPU devices. In general, serial codes that exhibit little or no data parallelism are executed on the host while parallel codes that exhibit rich amount of data parallelism are executed on the device. An entire procedure involves four steps: (1) Do initial jobs on host. (2) Copy input data form host to device. (3) Execute kernels on GPU. (4) Copy output data from device to host.

The most popular approach to realize GPU parallel computing is to decompose a problem into well-defined, thread-level work units, which are coded as kernels. Kernels are executed by a batch of threads, which has arranged by three-dimensional blocks. All threads in a block can communicate and synchronize with each other by synchronization functions. Furthermore, in order to manage the larger number of threads effectively, thread blocks are also managed by a three-dimension array, which is referred to as grids.

CUDA threads access data from a multiple device memory system to improve the memory bandwidth. All threads can access to the same global memory which is off-chip, big size and low bandwidth. Threads in a thread block share data through shared memory which has very high bandwidth and with the same lifetime as the block. Each thread has a private local memory and a set of registers. There are two additional read-only and off-chip memories accessible by all threads: the constant memory and texture memory.

Finally, although the CUDA affords a convenient tool to realize applications executing on GPU, the performance of the GPU-based application is difficult to be optimized. The performance with different execution configurations can be quite different, so there are a series of performance guidelines deserved to be observed when the CUDA is used. The details of optimization can be found in the official guide [30] and the corresponding literatures [31,32].

3.2. Basic parallel execution strategies

Compare to the implicit integration approach, the explicit integration approach has an excellent natural parallelism, because the most parts of the explicit application can be performed in their own data space independently. Therefore, each element or node can be seen as an independent thread-level work unit, it means that these variables should be parallelized randomly. According to these work units, three kinds of parallel execution strategies, including one thread map to one element (TME), one thread map to one node (TMN) and one thread map to one freedom (TMF) are suggested. For example, the nodal displacement vector can be calculated by mapping one thread to one node (TMN). In other words, each element of this vector is calculated by a different thread parallel. Overall, the GPU-based iteration application lets iteration steps span the horizontal direction, while parallelization spans the vertical direction, as shown in Fig. 5. In order to guarantee the stability and the conformity of these two span implementations, a memory operation or barrier synchronization often plugged between two parallel kernels or one parallel kernel and one non-parallel subroutine. The original two/multi-dimension arrays can usually be spitted into several one-dimension arrays in our GPU-based applications to ensure the coalescing accesses of global memory.

3.3. Parallel solver for single values using cuBLAS

In the process of FE analysis, many single values should be obtained among an array. These single values could be the sum, the maximum or the minima among all elements or all nodes, etc. Since reduction of the data transferred between the host and the device is very important to improve the efficiency of GPU-based applications, it is necessary to parallelize the calculations of these single values on GPU. Initially, a parallel reduction algorithm is introduced to our developed program, which uses a binary operation to reduce an input sequence to a single value and it has been effective implemented on GPU [33]. Nowadays, the CUDA basic linear algebra subroutines (cuBLAS) developed by NVIDIA

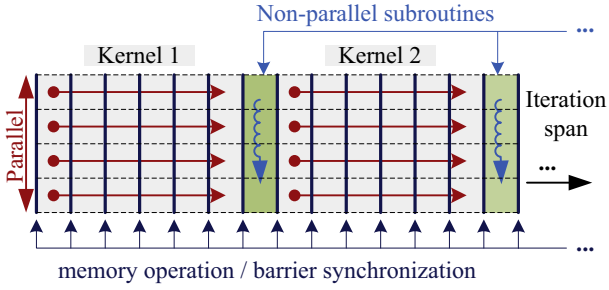


Fig. 5. GPU-based two span implementation.

makes it easy for researchers to code single value calculating program. Take the calculation of time-step size as an example, the Fig. 6 illustrates the way to obtain the minimum time-step size among all elements by function cublasIdamin() and cublasGetVector(), where *E* stands for the element information and *S* stands for the size of element. In the cuBLAS-based application, the areas array of each element is first parallelized by using the TME strategy. Sequentially, the function cublasIdamin() is used to find the index of the element with the minimum area. Then, the function cublasGetVector() copies the minimum value from the device memory to host memory. Finally, the time step size is calculated on CPU. Details are shown in Listing 1.

3.4. Parallel implementation of the element formulation

The way to parallel solve the element formulation has been discussed in our previous works [28,34], which presented various parallel FE algorithms based on central difference method and GPU general computing platform for plane nonlinear dynamic problems, such as an efficient parallel BT element solution. We can copy these parallel implementations to this parallel crashworthiness simulation system. Fortunately, the modular programming makes these works easy. After the primary parallel simulation system is developed, a thin-walled beam impact model with 4140 nodes and 4080 elements running on the Intel Core i7-930 with a single thread at 2.8 GHz and the NVIDIA GeForce GTX580 with 512 cores is considered to analyze the efficiency.

Because most of procedures in a single iteration step consume a short periods of time, we use the clock function clock() at the begin and the end of each procedure to get the number of clock cycles elapsed. For a GPU kernel, a function __syncthreads() is added at the end to synchronize the CPU and GPU. Fig. 7 shows the number of clock cycles consumed by each procedure of this partial parallel system for the beam model. It is easy to observe that the time consumed for the element formulation is obviously reduced. In the meantime, the contact algorithm has replaced the element formulation as the most time-consuming part. On the other hand, the overall efficiency of this partial system is not improved. Therefore, the efficiency of contact algorithm should be improved to obtain an obviously performance boost.

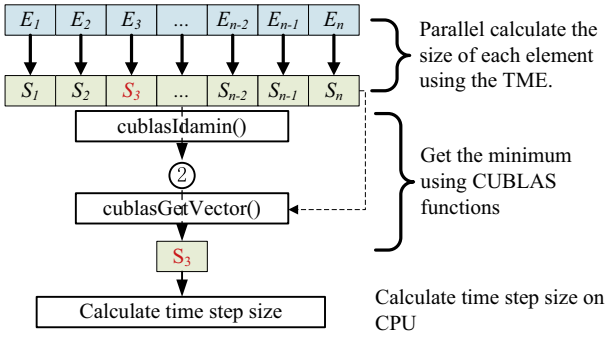


Fig. 6. Parallel algorithm for calculating time step using reduction.

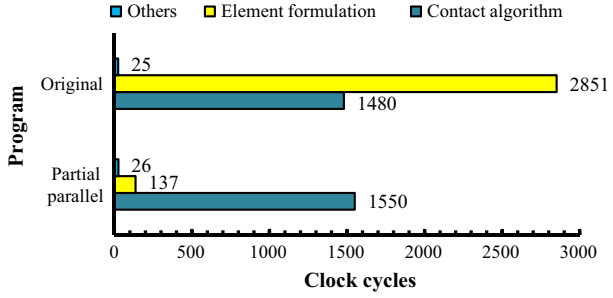


Fig. 7. Number of clock cycles consumed by the partial parallel system.

3.5. Parallel implementation of contact searching

Based on the above-mentioned theoretical analysis, the modules of the HITA method can be summarized to different function modules, as shown in Fig. 8. In this section, we will detail the parallel implementation of these modules. It should be noted that the hierarchy territories and the expanded territories are independent calculated for all segments, so they are easy to be parallelized by mapping one thread to one segment as same as the above-mentioned three basic parallel execution strategies.

3.5.1. Parallel test pairs searching

The module of the test pairs searching (module 3) needs to perform numerous of comparisons between the expanded territories and the segment nodes to find out which segment might be involved in a contact interfaces. For the case of three-dimension problems, our program stores the expanded territories and the segment nodes in a mixed array according to an optimal dimension of their coordinates. The optimal dimension usually stands for the direction in which has the largest number of segments. As shown in Fig. 9(a), the low bounds of expanded territory are stored in the front part of array and the low bounds of contact nodes are stored in the latter part. Then, the mixed array is sorted in ascending order as shown in Fig. 9(b), where the solid spot stands for the

Module 1	Pre-searching processing
	Loop over the number of time steps
Module 2	**Loop over** the number of contact nodes Calculate the territories **End Loop** over the number of nodes
Module 3	**Loop over** the number of contact nodes Find the test pairs by calculating the intersection of expanded territories **End Loop** over the number of nodes

Module 4	**IF** the number of test pairs is not 0 **THEN**
	Loop over the number of test pairs
	Find the contact pairs by calculating the exact position of a hitting node in relation to a relevant target hierarchy
	End Loop over the number of test pairs
	END IF

	End Loop over the number of time steps

Fig. 8. Modules of the HITA algorithm.

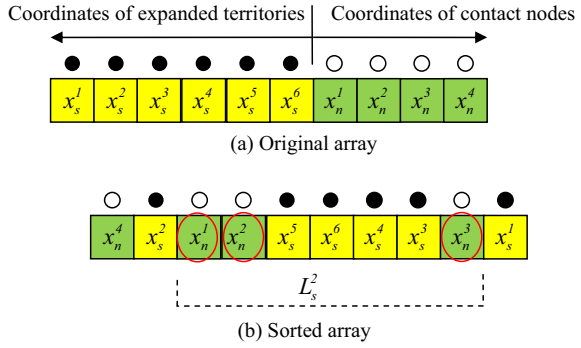


Fig. 9. Sorting low bounds before searching.

expended territories, and the vacant spot stands for the nodes of segment. Thereafter, the test pairs searching can be easily implemented by iterating over the sorted array. When an element of expanded territory is found, the program calculates the length of the expanded territory (L_s) in the optimal dimension and records the contact nodes falling in the area of L_s. For example, the nodes falling in the related area of expanded territory 2 are node 1, node 2 and 3. When the first searching in the optimal dimension is completed, a further comparison will be performed between the coordinates in the other two dimensions.

We developed several fine-grained parallel execution strategies to perform test pairs searching on GPU. The first step is the parallel construction of the mixed array by the TME strategy and stores it in the global memory. Secondly, this mixed array should be sorted in ascending order on GPU. Several parallel sorting methods based on the GPU have been published [35], including the above-mentioned parallel reduction algorithm. In order to decrease the parallel programming difficulty, our program chooses the Thrust [36] as the parallel sorting tool. Thrust is a free C++ template library for CUDA. Mixed array sorting can be easy achieved by means of the Thrust's sorting function thrust::sort_by_key() as shown in Listing 2.

In the third part, the way to traverse this sorted array involves a nest loop. At worst, the time complexity may up to $O(n^2)$, and it will consumes more than 90% of execution time of whole test pairs searching. The most direct approach to realize parallel traversal is to unroll the outer loop by mapping each array element to different threads. When a low bound is found by a thread, a more precise comparison is performed by this thread. This direct parallel strategy can reduce the time complexity to $O(n)$. Take a parallel execution model with 10 threads as an example. As shown in Fig. 10, in the beginning, 10 threads are assigned to parallel execute the $e1$ instruction to judge whether the current element is expended territory or not. If the result of one thread is FALSE, the program will terminates this thread while the rest threads continue execute the $e2$ instructions. The $e2$ instruction stands for the above-mention further comparison. If the result of the $e2$ instruction is TRUE, the program will records this test pair to global memory by the $c1$ instruction; otherwise, the program will terminates this thread. The storage location of each test pair is random according to the timestamp. For example, the test pair found by the thread 1 is stored in the location M1 and the one found by the thread 6 is stored in the location M2. Here an atomic add function atomicAdd() is used to avoid the race write.

3.5.2. Parallel contact pairs searching

In the HITA method, the approach to contact pair searches (module 4) in each hierarchy is to calculate the exacting distances between the contact node and its relevant target hierarchy. These

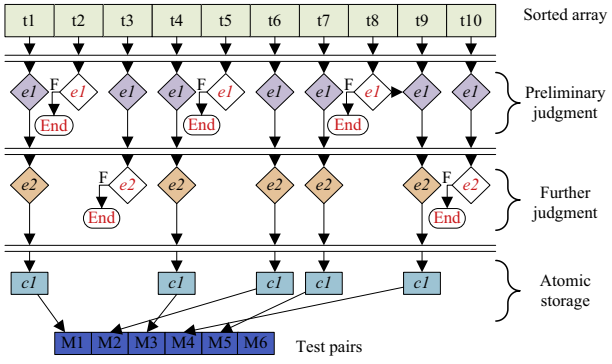


Fig. 10. Parallel strategy for test pairs searching.

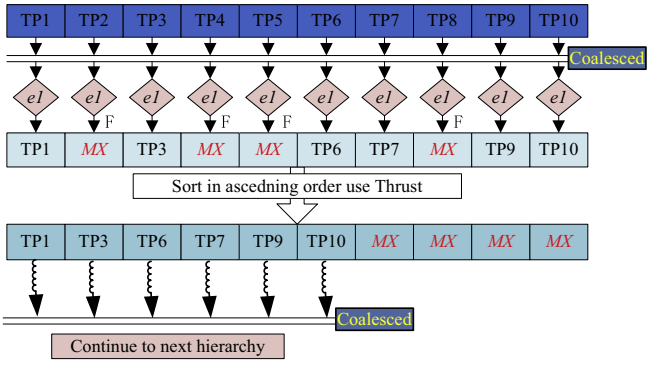


Fig. 11. Parallel strategy for contact pairs searching.

computations also give good fine-grained parallelism because of their data independence. To effective utilize the GPU's multi-processors, the suggested strategy takes one thread to calculate one test pair.

In this work, the test pairs in each hierarchy are stored in a same array to save the GPU memory consumption and reduce the time of allocating new memory space. Therefore, we performed a sorting operation when the searching in the higher level is completed; it is an effective strategy to ensure the updated array can be always coalesced accessed during the searching, as shown in Fig. 11. Take a 10 test pairs model as an example. Firstly, 10 threads are assigned to carry out a $e1$ judgment, $e1$ stands for the Eq. (1) for two-dimension problem and Eq. (6) for three-dimension problem. If one of these test pairs obtains the TRUE result, the test pair number will remains the same. Otherwise, if one of test pairs obtains the FALSE result, the test pair number will changes to a maximum integer value *MX*. Usually, the *MX* is equal to a positive number larger than the total number of test pairs in the highest level hierarchy. Secondly, we sort the array in ascending order to move the test pairs whose results are TRUE in this hierarchy to the front part, and the failed test pairs have been moved to the latter part. Finally, the cuBLAS function cublasIdamx() is introduced to count the total number of the TRUE test pairs by finding the smallest index of *MX* in the sorted array.

3.6. Parallel calculation of contact force

The first step of the explicit contact force calculation method is to obtain the penetrations, and then calculate the element contact forces by Eq. (9). The TME strategy can be used to parallel calculations of these penetrations and element contact force because of their inheriting natural parallelism. After that, similar to the element internal forces assembling, the element contact forces need to be assembled to the global nodal forces. As same as the nodal forces assembling, several elements parallel assemble their forces to one node may cause "race written" error. Fortunately, since the computing scale of contact forces assembling is much smaller than the element internal forces assembling, the atomic operation is introduced to assemble these forces without any additional work. In detailed, an atomic add function atomicAdd() for double float number is developed [37] as the Listing 3, due to the atomic add function provided by CUDA can only be used for the integer and long data types.

3.7. Overall GPU-based flowchart

Finally, based on the original flowchart as shown in Fig. 5, the flowchart of GPU-based parallel crashworthiness simulation system is presented. As shown in Fig. 12, subscript C means running

on CPU, subscript G means running on GPU and subscript S means asynchronous execution. The whole iterative procedure is accelerated by GPU in this system. Because the CUDA supports asynchronous transfers to overlap the host computation with the asynchronous data and device computation, asynchronous transfers are used to transfer intermediate results from the device to host in order to hide the time taken for I/O operation. Finally, a GPU-based software for contact–impact problem is developed based on a self-developed serial contact–impact simulation system DYSI3D [38].

4. Numerical examples

In order to analyze the precision and efficiency of our simulation system, three cases have been selected. The precision verification is carried out by comparing the calculation results between the CPU code and the GPU code for the same test model. On the other hand, the performances on GPU are measured including the data transfer time. The speedup is calculated by dividing the total CPU consumed time by the total GPU consumed time.

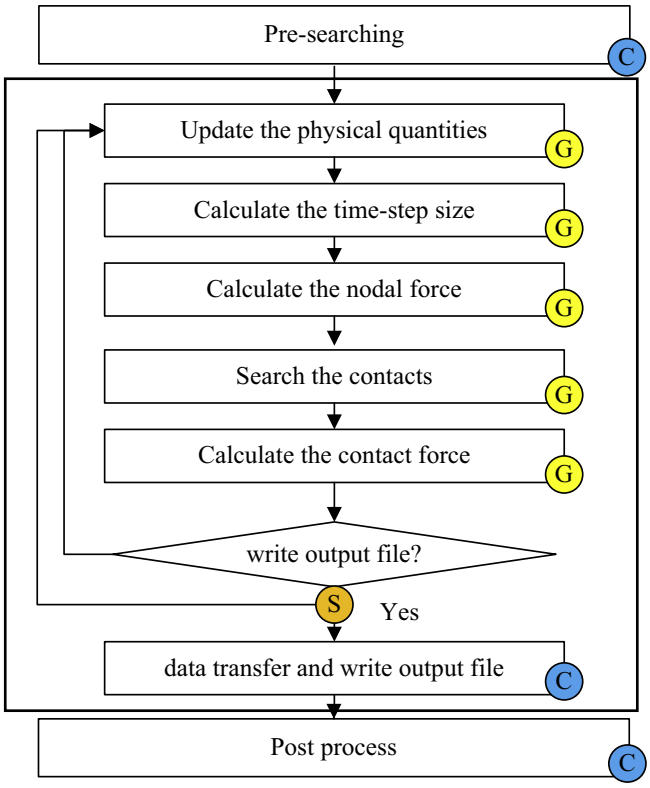


Fig. 12. Flowchart of the GPU-based parallel simulation system.

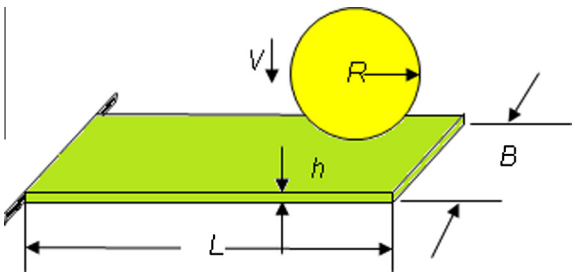


Fig. 13. A ball impacting a clamped beam.

All examples have been carried out on an Intel Core i7-930 with one single thread at 2.8 GHz. The GPU used in our tests is a NVIDIA GeForce GTX580, which including 512 CUDA cores and 3 Gb global memory. Furthermore, the version of CUDA for coding and compiling is 5.0. Float numbers in our program are all in double precision format.

4.1. Test case 1: a ball impacting a clamped beam

In this case, a ball impacting a clamped beam is studied to analyze the correctness of the suggested parallel contact searching strategy, as shown in Fig. 13. The details of this case can be found

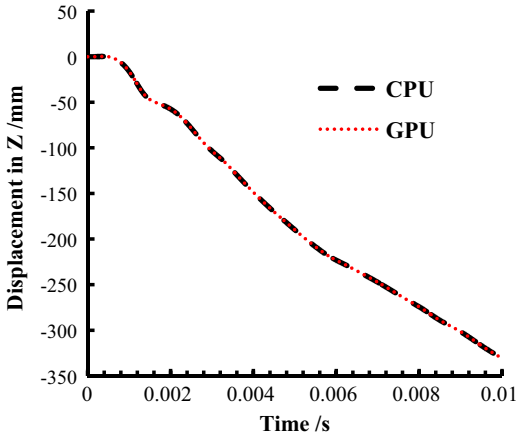


Fig. 14. Displacement time histories of one of the node in ball.

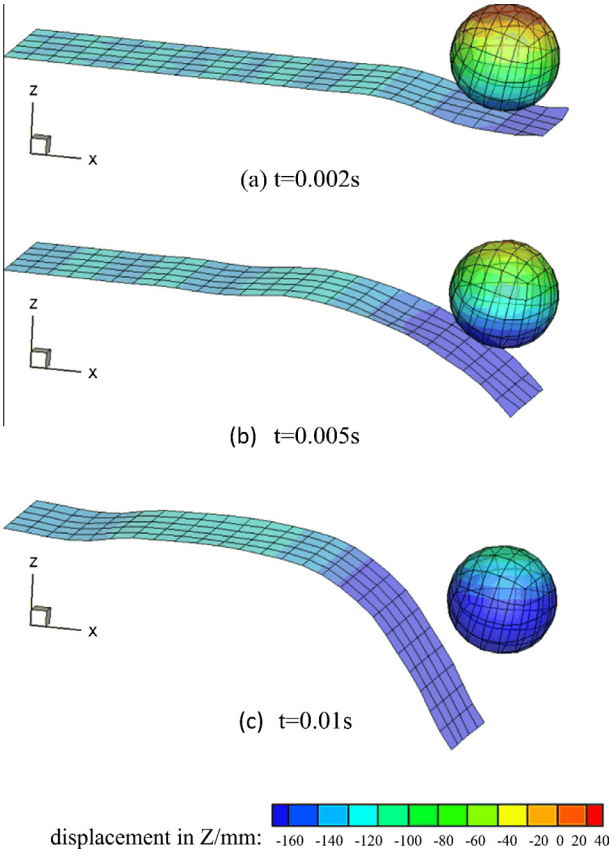


Fig. 15. Deformed configurations of the ball-beam system.

in [13], and the geometric of this model is defined with the parameters $R = 0.01$ m, $B = 0.24$ m, $L = 1.0$ m, $h = 0.0015$ m. Both the ball and beam are assumed to consist of elastic material with Young's module $E = 200$ GPa, and Poisson's ratio $v = 0.3$. Squared unstructured meshes are processed, and an initial velocity of 30 m/s is assigned to the ball.

The CPU and GPU calculated displacement–time curves of one of the nodes in the ball from 0 s to 0.01 s are plotted in Fig. 14. As can be seen, the result curve of CPU plotted in black dash line is the same as the result curve of GPU plotted in red dotted line. This phenomenon indicates the simulation results of GPU are identical to CPU. Meanwhile, three deformed shapes with displacement distribution in the Z direction of this model calculated by GPU are plotted in Fig. 15. It shows that there are several oscillations occurring in the displacement of the beam due to the mechanics feature of elastic material. Therefore, it can be concluded that the GPU codes simulate the contact–impact process as same as the original CPU code exactly.

4.2. Test case 2: a square beam impacting a rigid wall

In the second test case, a square shell beam impacting a rigid wall under a crashing load is considered. Such a square beam is typically used in the passenger car to absorb energy during a crash. The geometry parameters are shown in Fig. 16, and the elastic–plastic material parameters are as follows, Young's module $E = 0.21$E6 MPa, Poisson's ratio $v = 0.3$, Yield stress $\sigma_y = 206$ MPa, and density is 2.45E−6 kg/mm^3. The beam is assigned a linear initial velocity of 100 m/s impacting a fixed rigid wall from 0 s to 10 s.

The prepared FE model of this square beam has 24,840 freedom degrees and 4080 elements. The deformed plots with displacement distribution in the X direction of the beam at four different time steps are given in Fig. 17. We can see that the deformed shapes calculated by GPU are exactly as same as the deformed shapes calculated by CPU. This simulation consumed 4331.2 s with 1328 time steps on CPU while the elapsed time is reduced to 658.2 s with the same time steps on GPU. So, the speedup ratio of the GPU implementation for this simulation is 6.58.

4.3. Test case 3: an automobile body-in-white crash simulation

Finally, a crashworthiness simulation of an automobile BIW model is considered, as shown in Fig. 18. The elastic–plastic material parameters are as follows: Young's module $E = 0.21$E6 MPa, Poisson's ratio $v = 0.3$, Yield stress $\sigma_y = 206$ MPa, and density is 2.45E−6 kg/mm^3. The car is assigned an initial velocity of 16 m/s impacting a fixed rigid wall continuously for 0.05 s.

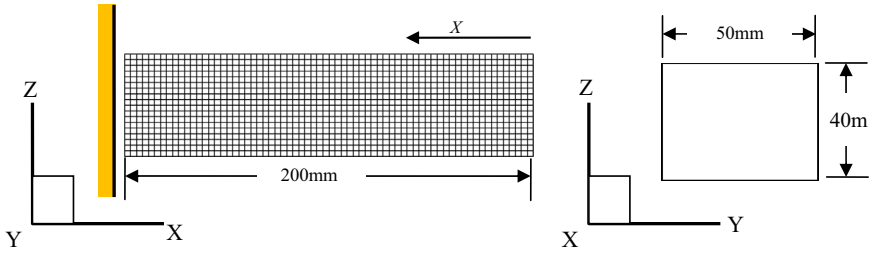


Fig. 16. A square beam impacting a rigid wall.

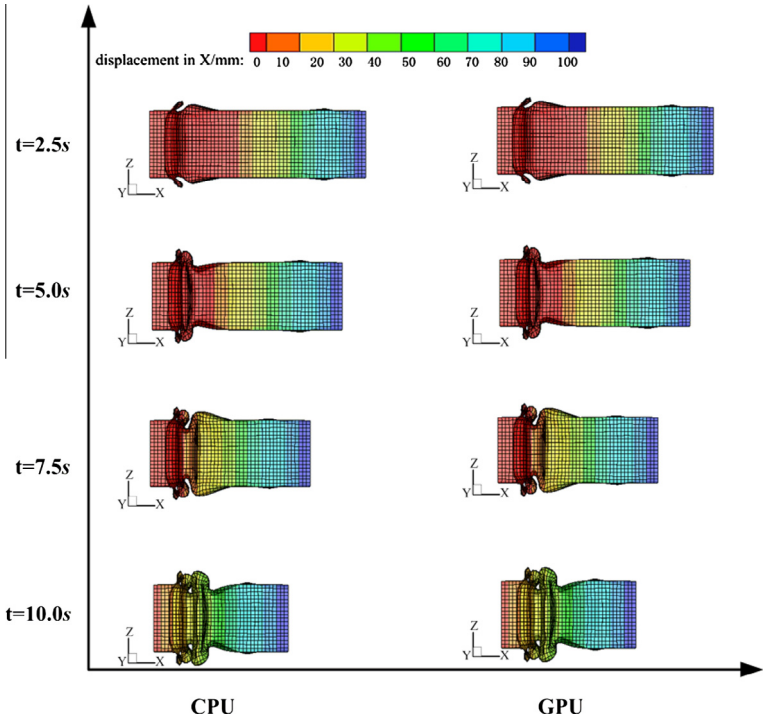


Fig. 17. Plots of deformation history of square beam.

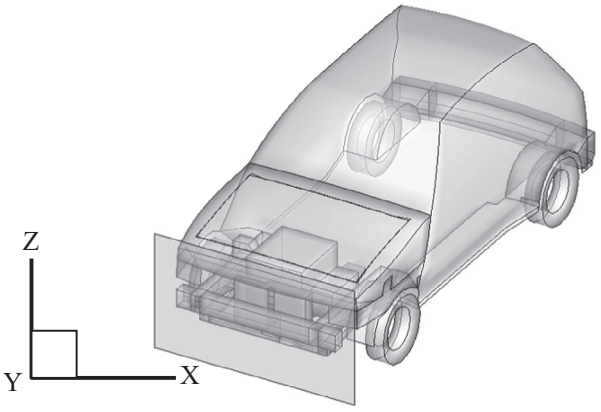


Fig. 18. Automobile body-in-white crash model.

Table 2
Three different scale finite element models.

	Elements	Nodes	Degrees
Case 1	20,597	22,404	134,424
Case 2	73,680	73,727	442,362
Case 3	294,720	294,803	1,768,818

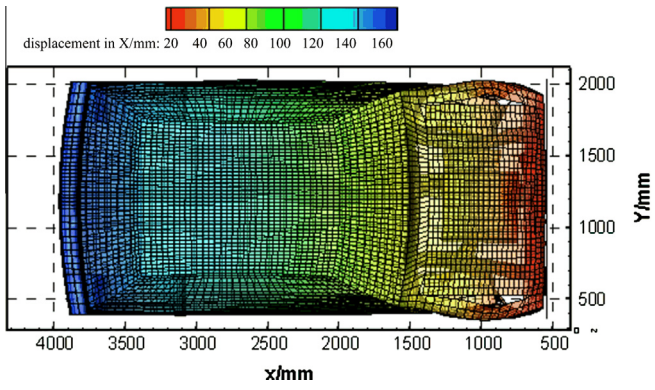


Fig. 19. Deformed BIW of model 2.

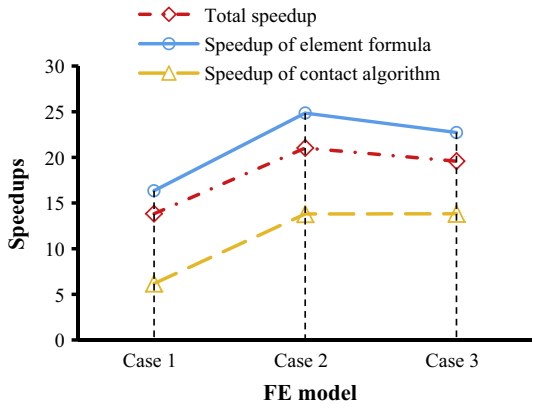


Fig. 20. Speedup ratios of three different models.

In order to verify the efficiency of GPU code for different computation scales, three different size meshes of this BIW model are considered, as shown in Table 2. These three FE models are all simulated on both CPU and GPU. Fig. 19 shows the GPU calculated deformed body with the displacements distribution in the *X* direction of the case 2 model, where 143,028 time steps in total are calculated to simulate the 0.05 s impact process.

Take the case 2 model for example, the speedups of element formula and contact algorithm than run on GPU compared to its CPU counterpart is shown in Fig. 20. It can also be seen in Fig. 20, the maximum speedup of the GPU-based simulation system tends to 22, compare to running on the CPU with one single thread.

```
1  ...
2  extern "C"
3  void TimeStepGPU(intgmeshl, intgnnode, intndf, int *d_ix, double
4  *d_x,double&gdt1,double &gdt2, double gyms, double gro)
5  {
6  //Calculate the area of element
7  CalculateArea<<<numBlocksTS,numThreadsTS>>>(gmeshl,gnnode,ndf,d_ix,
8  d_x,d_gcb);
9  // Find the minimum area
10 // Find the index of the element of the minimum area
11 cublasIdamin(handle1,gmeshl,d_gcb,1,&idex);
12 // Get the value by the index
13 cublasGetVector(1,sizeof(double),&d_gcb[idex-1],1,&tempmin,1);
14 // Calculate the time stepsize
15 double wv = sqrt(gyms/gro);
16 gdt2 = gdt1;
17 gdt1 = 0.9 * tempmin / wv;
18 }
```

Listing 1. Calculation of the time-step size using cuBLAS.

```
1  …
2  // Include Thrusthead files
3  #include <thusrt/sort.h>;
4  #include <thusrt/device_ptr.h>;
5  …
6  // Construct the mixing array: d_xsgd.
7  // The size of array is the number of contact nodes add the number of segments.
8  // Setup an index array: d_ksgd.
9  // The elements of d_ksgdared_xsgd corresponding node number or segment number.
10 //sort this mixing array in ascending order
11 thrust::sort_by_key(thrust::device_ptr<double>(d_xsgd),
12 thrust::device_ptr<double>(d_xsgd+nums),
13 thrust::deivce_ptr<int>(d_ksgd));
14 …
```

Listing 2. Sort the mixing array by the Thrust.

```
1  __device__ double atomicAdd(double* address, double val)
2  {
3  unsigned long longint* address_as_ull = (unsigned long longint*)address;
4  unsigned long longint old = *address_as_ull, assumed;
5  do {
6  assumed = old;
7  old = atomicCAS(address_as_ull, assumed,
8  __double_as_longlong(val +
9  __longlong_as_double(assumed)));
10 } while (assumed != old);
11 return __longlong_as_double(old);
12 }
```

Listing 3. Atomic add function for double float.

5. Concluding remarks

In this study, a parallel crashworthiness simulation system based on GPU is presented. In the beginning, three kinds of GPU-based parallel approaches for explicit FE algorithm are presented, including TME, TMN and TMF. These suggested approaches inherited the good parallelism of explicit FE algorithm and they can be efficiently executed on GPU. Furthermore, two free GPU-based numerical calculation libraries, include cuBLAS and Thrust, are introduced into our system to decrease the difficulty of programming. In order to analyze the time distribution of the original sequential simulation system, a BIW crashworthiness simulation model with 69,625 nodes and 65,177 elements that runs on CPU was studied. It shows that the contact searching is the most time-consuming part, while the element formula is the second time-consuming part. According to our previous research work, a GPU-based parallel element formula solver with BT shell element is copied to the original sequential system DYSI3D to form a partial parallel system. The numerical result of this primary system shows that the time consumed to solve element formula was significantly decreased, but the whole efficiency was barely improved. Therefore, an entire parallel contact searching algorithm based on the HITA method is presented. During the parallel test pairs searching, a mixed array is allocated in global memory and parallel calculated by the TME strategy to improve the searching efficiency. Since the test pairs searching need a two-lever loop to traverse this mixed array, the outer loop is unrolled to realize parallel searching. Next, as the contact pairs searching gives good finer-grained parallelism, one CUDA thread is assigned to judge one test pair to realize parallel searching. Furthermore, atomic function is introduced to record search results and a sort-based strategy is presented to translate the search result between different hierarchies. Finally, the approach to contact force parallelization is discussed. The TME pattern is adopted to calculate the penetrations and the penetration forces. A double float atomic add function is developed for contact forces assembling.

Three kinds of classical impact–contact problems are investigated as the test cases for the GPU-based simulation system. All examples have been run on the Intel Core i7-930 with a single thread and the NVIDIA GeForce GTX580 with 512 cores. By using double precision floating point numbers, the parallel algorithm running on GPU can obtain the same results as the CPU-based implementation. Take the case 2 of BIW models as an example, this parallel implementation can perform more than 22 times faster on CPU. This means that the GPU-based simulation system can reduce the calculation time remarkably and save calculation cost. It makes possible to mesh more elements during the simulation, which can better represents the geometry and provides more accurate results.

Acknowledgment

This work has been supported by Project of the Key Program of National Natural Science Foundation of China under the Grand Number 61232014.

References

[1] Bulik M, Liefvendahl M, Stocki R, Wauquiez C. Stochastic simulation for crashworthiness. Adv Eng Softw 2004;35:791–803.
[2] Wang H, Li GY, Li E. Time-based metamodeling technique for vehicle crashworthiness optimization. Comput Methods Appl Mech Eng 2010;199:2497–509.
[3] Pifko A, Winter R. Theory and application of finite element analysis to structural crash simulation. Comput Struct 1981;13:277–85.
[4] Johnson J, Skynar M. Automotive crash analysis using the explicit integration finite element method. ASME Publisher, AMD; 1989.
[5] Hughes TJ, Hinton E. Finite element methods for plate and shell structures. Pineridge Press International; 1986.
[6] Hughes TJ, Liu WK. Nonlinear finite element analysis of shells: Part I. Three-dimensional shells. Comput Methods Appl Mech Eng 1981;26:331–62.
[7] Hughes T, Liu W, Levit I. Nonlinear dynamic finite element analysis of shells. Nonlinear Finite Elem Anal Struct Mech 1981:151–68. Springer.
[8] Belytschko T, Lin JI, Chen-Shyh T. Explicit algorithms for the nonlinear dynamics of shells. Comput Methods Appl Mech Eng 1984;42:225–51.
[9] Belytschko T, Leviathan I. Physical stabilization of the 4-node shell element with one point quadrature. Comput Methods Appl Mech Eng 1994;113:321–50.
[10] Belytschko T, Wong BL, Chiang H-Y. Advances in one-point quadrature shell elements. Comput Methods Appl Mech Eng 1992;96:93–107.
[11] Hughes TJR, Taylor RL, Sackman JL, Curnier A, Kanoknukulchai W. A finite element method for a class of contact–impact problems. Comput Methods Appl Mech Eng 1976;8:249–76.
[12] Belytschko T, Neal MO. Contact–impact by the pinball algorithm with penalty and Lagrangian methods. Int J Numer Methods Eng 1991;31:547–72.
[13] Zhong Z-H. Finite element procedures for contact–impact problems. Oxford: Oxford University Press; 1993.
[14] Fahmy M, Namini A. A survey of parallel nonlinear dynamic analysis methodologies. Comput Struct 1994;53:1033–43.
[15] Namburu RR, Turner DA, Tamma KK. An effective data parallel self-starting explicit methodology for computational structural dynamics on the connection machine CM-5. Int J Numer Methods Eng 1995;38:3211–26.
[16] Plaskacz EJ, Belytschko T, Chiang H-Y. Contact–impact simulations on massively parallel SIMD supercomputers. Comput Syst Eng 1992;3:347–55.
[17] Belytschko T, Plaskacz E, Chiang H-Y. Explicit finite element methods with contact–impact on SIMD computers. Comput Syst Eng 1991;2:269–76.
[18] Namburu RR, Turner DA. Contact impact algorithm on a data parallel computer. High-performance computing in computational dynamics. New York: ASME; 1994. p. 111–9.
[19] Paik SH, Moon JJ, Kim SJ, Lee M. Parallel performance of large scale impact simulations on Linux cluster super computer. Comput Struct 2006;84:732–41.
[20] Malone JG, Johnson NL. A parallel finite element contact/impact algorithm for non-linear explicit transient analysis: Part II—parallel implementation. Int J Numer Methods Eng 1994;37:591–603.
[21] Brown K, Attaway S, Plimpton S, Hendrickson B. Parallel strategies for crash and impact simulations. Comput Methods Appl Mech Eng 2000;184:375–90.
[22] Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Buijssen SH, Grajewski M, et al. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Comput 2007;33:685–99.
[23] Mafi R, Sirouspour S. GPU-based acceleration of computations in nonlinear finite element deformation analysis. Int J Numer Methods Biomed Eng 2014;30:365–81.
[24] Ikushima K, Shibahara M. Prediction of residual stresses in multi-pass welded joint using idealized explicit FEM accelerated by a GPU. Comput Mater Sci 2014;93:62–7.
[25] Komatitsch D. Fluid–solid coupling on a cluster of GPU graphics cards for seismic wave propagation. C R Mécanique 2011;339:125–35.
[26] Georgescu S, Chow P, Okuda H. GPU acceleration for FEM-based structural analysis. Arch Comput Methods Eng 2013;20:111–21.
[27] Banaś K, Płaszewski P, Macioł P. Numerical integration on GPUs for higher order finite elements. Comput Math Appl 2014;67:1319–44.
[28] Cai Y, Li G, Wang H, Zheng G, Lin S. Development of parallel explicit finite element sheet forming simulation system based on GPU architecture. Adv Eng Softw 2012;45:370–9.
[29] Hallquist J. NIKE2D: an implicit, finite-deformation, finite-element code for analyzing the static and dynamic response of two-dimensional solids. California Univ., Livermore (USA). Lawrence Livermore Lab.; 1979.
[30] Nvidia C. C best practices guide. Santa Clara, CA: NVIDIA; 2012.
[31] Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Skadron K. A performance study of general-purpose applications on graphics processors using CUDA. J Parallel Distrib Comput 2008;68:1370–80.
[32] Ryoo S, Rodrigues CI, Stone SS, Stratton JA, Ueng S-Z, Baghsorkhi SS, et al. Program optimization carving for GPU computing. J Parallel Distrib Comput 2008;68:1389–401.
[33] Kirk DB, Wen-mei WH. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann; 2010.
[34] Yong C, Guangyao L, Hu W. Parallel computing of central difference explicit finite element based on GPU general computing platform. J Comput Res Dev 2013;50:412–9.
[35] Sintorn E, Assarsson U. Fast parallel GPU-sorting using a hybrid algorithm. J Parallel Distrib Comput 2008;68:1381–8.
[36] Bell N, Hoberock J. Thrust: a productivity-oriented library for CUDA. GPU Comput Gems 2011:7.
[37] Nvidia C. Programming guide; 2008.
[38] Xie H, Zhong Z, Li G, Cheng A, Cao Y. Parallel computation and application of sheet forming numerical simulation. Zhongguo Jixie Gongcheng/China Mech Eng 2003;14:1842–4.