Architecting the Discontinuous Deformation Analysis Method Pipeline on the GPU

Yunfan Xiao, Min Huang, Qinghai Miao*, Jun Xiao and Ying Wang
School of Engineering Science
University of Chinese Academy of Science
Beijing, China
E-mail: xiaoyunfan12@mails.ucas.ac.cn {huangm, miaoqh, xiaojun, ywang}@ucas.ac.cn

***Abstract*—As an important numerical analysis method of rock mechanics, discontinuous deformation analysis (DDA) has been widely used in rock engineering. DDA has certain advantages such as the large time step and the large deformation, at the cost of relatively low computing efficiency. To address the efficiency bottleneck of DDA, this paper proposes a complete graphics processing unit (GPU)-based version. The entire DDA pipeline, involving contact detection, global matrix building, linear equation solving, and interpenetration checking, is restructured according to the GPU architecture to minimize data transmissions between the host and device. For the equation solver in DDA, a comparison study of the conjugate gradient method with different preconditioners, i.e., block Jacobi, symmetric successive over-relaxation (SSOR) approximate inverse, and ILU, is introduced first, and a novel sparse matrix-vector multiplication (SpMV) method, intended for the sparse block symmetry matrix with distinct features and which outperforms cuSPARSE by 2.8 times, is proposed as well. Schemes to solve memory write conflicts and branch divergences on the GPU are also introduced in contact detection, global matrix building, and interpenetration checking. For the stable analysis of a slope, the proposed GPU-based DDA with double precision achieved a speed-up rate that was 48.72 times higher than that of the original CPU-based serial implementation.**

Keywords-Discontinuous Deformation Analysis, Graphics Processing Unit (GPU), Sparse Matrix-vector Multiplication (SpMV), Contact Detection, Memory Write Conflict, Branch Divergence

I. INTRODUCTION

Discontinuous deformation analysis (DDA) is a type of discrete element method originally proposed by Shi in 1988. It can analyze the mechanical response of blocky systems under general loading and boundary conditions. Large displacements and deformations are considered under both static and dynamic loadings [1].

DDA simulates the interaction of discrete bodies with multi-time steps as discrete element method (DEM), but solves stress-displacement problems in each step as finite element method (FEM). DDA derives the global simultaneous equations from the minimum potential energy principle and takes displacements as unknowns, which are solved in each iterative step. On the other hand, DDA gives a real dynamic solution with the correct energy consumption by frictional resistance at contact and at the velocities passing between the successive steps [2].

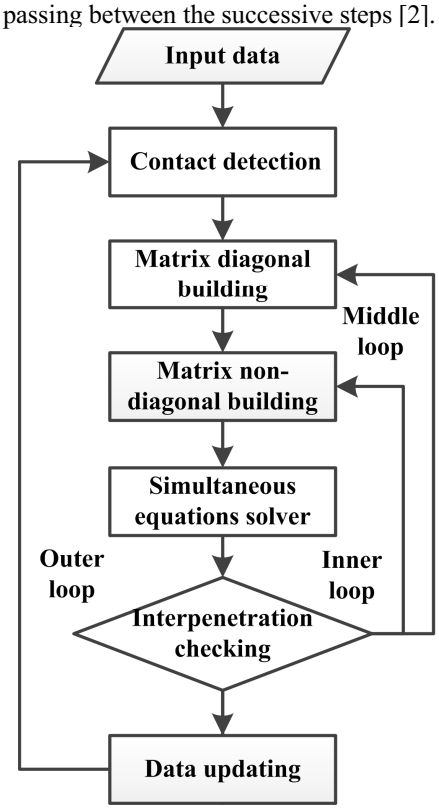

Figure 1. The pipeline of DDA on the CPU

These advantages of DDA lead to massive computation. Besides data input, the pipeline of DDA mainly includes six computing modules, including the contact detection module, global stiffness matrix diagonal building module, global stiffness matrix non-diagonal building module, sparse linear symmetry equation solving module, interpenetration checking module, and data updating module. As illustrated in Fig. 1, the iterative loop in the pipeline has three nested loops. The outer loop (loop 1) is the multi-time iterative step. The results of the previous step will be the input data of the next step. This iterative step allows DDA to simulate the large displacement and deformation in blocky systems. The middle loop (loop 2) is the maximum allowed displacement iterative process. The displacement of each block in the current step must be less than the double of the maximum allowed displacement, which is a control

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.93

parameter in DDA to ensure the assumption of infinitesimal step displacements [3]. The inner loop (loop 3) is the open-close iterative process. It checks the contact and stress states among all the blocks to ensure that there are no interpenetrations between the contacted blocks and no tension between the separate blocks. The deeper the level these modules are in, the more time they consume in serial computing. However, contact detection is an exception, which only consumes less time than that by the sparse linear symmetry equation solving module. This three-level iterative pipeline with frequent contact detection and equation solving limits the efficiency in practical applications.

In recent years, the graphics processing unit (GPU) has occupied a strong position in high-performance computing owing to its high computing capacity at low cost. The thousands of arithmetic logical units (ALUs) in one chip and the wide memory bandwidth enable it to perform arithmetic operations in bulk and demonstrate dramatic peak performance in the computation, especially for large-scale data set processes with simple control and logical operations [4].

However, the single-instruction multiple-thread (SIMT) architecture and hierarchical memory model make it difficult for numerical algorithms to exploit the full capacity of a GPU directly. On the other hand, the memory bandwidth of a GPU is still a bottleneck compared to the cores' compute capacity. For example, for an advanced GPU with double precision such as the Tesla K40, the peak performance of double-precision floating-point operations can reach 1.43 Tflops/s, whereas that of single-precision floating-point operations can reach 4.29 Tflops/s, but the memory bandwidth is 288 GB/s, which means that the peak performance can only be acquired if the ratio between an arithmetic operation and the memory access is more than 36(1.43×8×(10^12)/288×(1024^3)) times for the double-precision floating-point operation and 55(4.29×4×(10^12)/288×(1024^3)) times for the single-precision one. For the full utilization of the GPU, considerable efforts at algorithm rebuilding and optimization are required.

Generally, architecting the pipeline of DDA to achieve outstanding performance on the GPU is potentially challenging. The contributions made by this work are highlighted as follows:

- Reconstruction of the framework of DDA on a GPU is performed. All the computing models are transplanted to execute on a GPU to obtain an obvious speed-up. The framework is based on data classification to reduce the branch divergence on a GPU.
- A comparison study of the preconditioners of the conjugate gradient (CG) method in DDA is made, which provides guidance in choosing an optimum preconditioner for it.
- A novel SpMV method with a matrix storage format named half slice block compressed sparse row format (HSBCSR) is proposed. It takes advantage of the natural blockiness and symmetry of the global matrix in DDA to achieve better performance compared to other methods.
- Schemes are proposed to solve the memory write conflicts and branch divergence problems on the GPU in the implementation of contact detection, global stiffness matrix building, and interpenetration checking.

II. RELATED WORKS

In the past decades, extensive studies on executing related algorithms on the GPU have been reported in FEM and DEM, especially the contact detection and linear equation solving. The computing modules of DEM are naturally suited for the GPU. They had been realized on the GPU and had acquired a noteworthy speed-up early on [5]. Later, a GPU-based coupled FEM/DEM procedure was proposed, which achieved a multi-hundred-fold increase in speed-up rate during that period [6]. The whole pipeline of FEM was also architected on the GPU, and an algebraic multigrid (AMG) preconditioned conjugate gradient (PCG) achieved an increase in speed-up rate by dozens of times in the equation solving [7]. However, limited studies are reported in the parallelization of DDA. A parallel method for solving simultaneous equations using a numerical manifold method on the CPU was reported in [8]. OpenMP-based parallel Jacobi, block Jacobi (BJ), and symmetric successive over-relaxation (SSOR) PCG methods for solving simultaneous equations in DDA were reported in [9]. In [10], a hybrid CPU-GPU-based DDA with contact detection, equation solving, and interpenetration checking on a GPU was reported; however, the massive data transmission between the CPU and the GPU limited the speed-up rate by 2 to 10 times.

For specific algorithms, two relevant issues, namely, contact detection and sparse linear symmetry equation solving, are briefly reviewed.

A. Contact Detection on the GPU

Real-time contact detection on the GPU has been a research hotspot derived from computer graphics. Most of the studies focus on broad phase detection. The basic method called brute-force collision detection (BFCD) has a complexity of $O(n^2)$. "Sort and sweep" and "spatial subdivision" techniques decreased the complexity to $O(n)$ or $O(nlogn)$ [11]. Construction, updating, and traversal of a tight-fitting bounding volume on the GPU were performed in [12]. In [13], a bounding volume test tree was performed on GPU clusters with collision-packet traversal and workload balancing scheme to boost the efficiency of parallel processing. In previous studies, hierarchical grids and octree-based strategies for spatial subdivision were compared [14], and a hash grid-based spatial subdivision was performed in the broad phase detection of DEM [15]. However, all these methods need extra preconditions, which could be time consuming and complex to execute on the GPU. The method used here was an improvement of that used in [16], as it ensures high efficiency by load balancing and effective cache utilization with a simple precondition on the GPU [10].

B. *Sparse Linear Symmetry Equation Solving on the GPU*

The PCG method has been the most commonly used solver for particular systems of linear equations, whose matrix is symmetric and positive definite. It belongs to the Krylov subspace iterative method and can be viewed as a direct method in theory since it produces the exact solution after a finite number of iterations. The low data dependency in the computation makes this method popular on multicore devices. However, two obstacles are still inevitable for implementing it on the GPU.

1) Preconditioner construction and implementation: In serial computing, incomplete LU decomposition (ILU), incomplete Cholesky decomposition (IC), and SSOR are good choices in most situations. Thus, a number of studies on how to execute them on the GPU have been reported [17]. However, the high data dependency of triangular equations solving has limited their efficiency [18]. The studies on parallel tridiagonal solvers are focused on multicore CPU devices, where these solvers have achieved better performance than their GPU-based counterparts [19]. BJ and Jacobi methods are easy to construct and implement on the GPU, but they have a low convergence rate with an ill-conditioned matrix [20]. Recently, AMG-PCG on the GPU is coming into focus owing to its low data dependency and high convergence rate. The details on how to construct and implement it on the GPU were reported in [21]. A multi-color variant of the symmetric Gauss-Seidel algorithm for the smoothing of AMG on the GPU was reported in [22]. However, the cost of constructing and implementing it are massive. Moreover, sparse approximate inverse and polynomial preconditioners on the GPU have also been reported [23].

2) Sparse matrix-vector multiplication (SpMV): The irregular access to vector and unbalanced loads of each row leads to low performance on the GPU. At first, basic schemes of SpMV on the GPU for different sparse matrix storage formats were proposed in [24]. The ELLPACK format (ELL) stands out since it is more robust than the diagonal format (DIA) and has better memory access pattern than other formats. It has been continuously improved to ELLPACK-R [25], sliced ELLPACK [26], ELLWARP [27], etc. ELL sorts the row of a matrix by the number of nonzero entries and then segments it into pieces to reduce the zero-filled data and keep load balance on the GPU. Moreover, the block compressed sparse row (BCSR) format is preferred in a block sparse matrix [28], and a scheme to solve the load balance problem and expose the parallelism of SpMV was proposed in [29].

The global stiffness matrix of DDA is naturally blocky and symmetric. A novel SpMV method that makes the full use of these features to achieve better performance is proposed here.

III. INTEGRATION OF THE DDA PIPELINE ON THE GPU

A. *Framework*

The framework of DDA on the GPU is restructured on the basis of data classification. As shown in Fig. 2, divergent data go through their own pipelines in the contact detection module, global stiffness matrix non-diagonal building module, and interpenetration checking module.

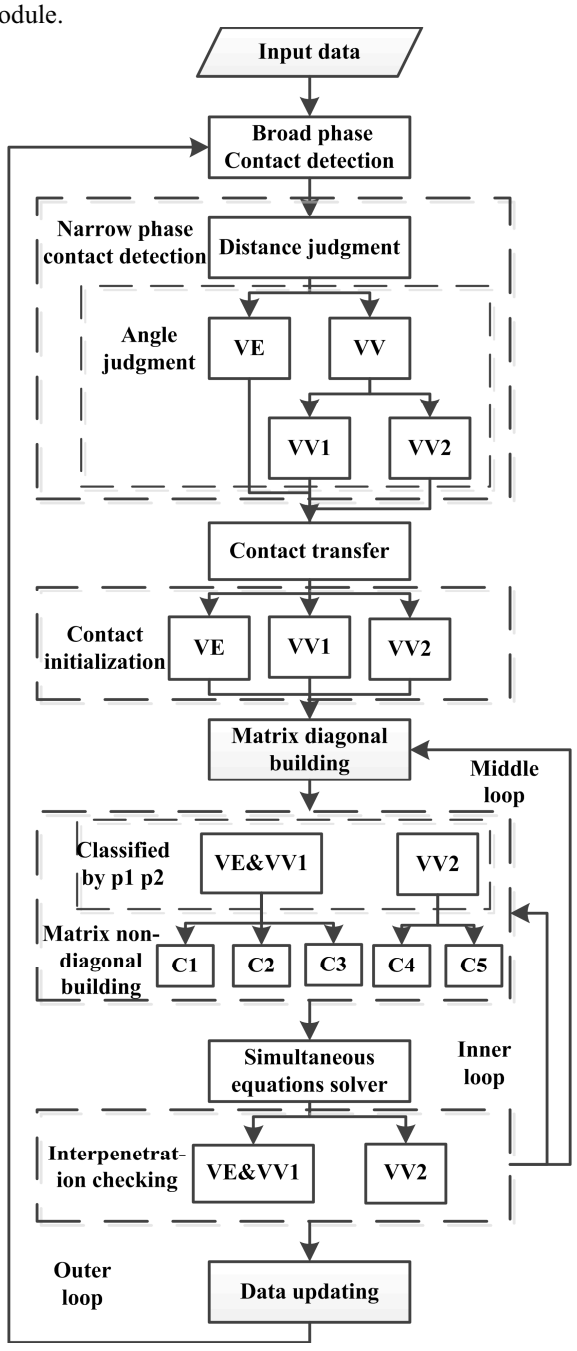

Figure 2. The pipeline of DDA on the GPU

The first classification occurs in the distance judgment step of the narrow phase of contact detection. According to the judgment results, the data without contact possibility are abandoned and the others are divided into vertex-edge (VE) and vertex-vertex (VV).

The second classification occurs in the angle judgment step of the narrow phase of contact detection. Some data of VE and VV are abandoned if they cannot fulfill the contact angle condition. The other data of VV will be divided into vertex-vertex-1 (VV1) and vertex-vertex-2 (VV2). The contacts with parallel edges are classified as VV1 or as VV2 otherwise.

The third classification occurs in the global stiffness matrix non-diagonal building. There are three contact models, namely, open, slide, and lock [1]. The switches of the contact model from the previous step to the current are indicated by two integers: p_1 and p_2. Their values are -1, 0, or 1. VE and VV1 are divided into three categories by p_1 and p_2. Category 1 (C1) is $|p_1|>0$. Category 2 (C2) is $|p_1|=0$ & $|p_2|>0$. Category 3 (C3) is $|p_1|=0$ & $|p_2|=0$ & *some other condition*. VV2 is divided into two categories based on p_1 and p_2. Category 4 (C4) is $|p_1|>0$. Category 5 (C5) is $|p_1|=0$ & $|p_2|>0$. If the contact data do not belong to any of the categories above, they will be abandoned.

In the interpenetration checking module, VE and VV1 will go through the same pipeline. VV2 will be computed individually.

All the data classifications are executed on the GPU. An efficient scan method [30] and radix sort method [31] were adopted to classify these data. The reduction algorithms in the scan and radix sort methods were replaced by a shuffle instruction [32] to achieve better performance on the GPU.

These computations among the different categories are individual. The branch divergence on the GPU can be reduced by the data classification. A case analysis shows that the data classification saves 20.576 us and reduces 11.18% branch divergence in the process of contact initialization, which is tested by Nsight[33].

B. Contact Detection

The components of contact detection in DDA include broad phase detection, narrow phase detection, contact transfer, and contact initialization.

In broad phase detection, the workflow is modeled as a matrix that operates on a vector. In serial computing, the matrix is an $n\times n$ upper triangular matrix. When mapping it to the GPU, it is reshaped as an $n\times(n/2)$ full matrix to ensure load balance. Then, dividing the matrix and vector into several $m\times m$ sub-matrixes and $m\times 1$ sub-vectors, the operation between each sub-matrix and sub-vector is mapped to one CUDA (Compute Unified Device Architecture) block. Only $2m-1$ entries are different in each $m\times m$ sub-matrix. They are stored in a shared memory for multiple access. The details of this part are reported in our previous work [10].

In narrow phase detection, the distances between the vertexes and the edges of each block couple that may have contact are computed. If their distances are less than a threshold, they will be recorded and classified as VE or VV. Then, their contact angles are determined to confirm whether the contact will happen or not, and the existing contacts are classified as VE, VV1, or VV2. Fig. 3 illustrates three typical contact models with their contact data. The contact data include contact vertex *i* and contacted edge *j1–j2* or vertex *j, j2*. They belong to two different blocks: *b1* and *b2*. Scan and radix sort methods are used here to classify or abandon data. Valid data will be stored in a successive array.

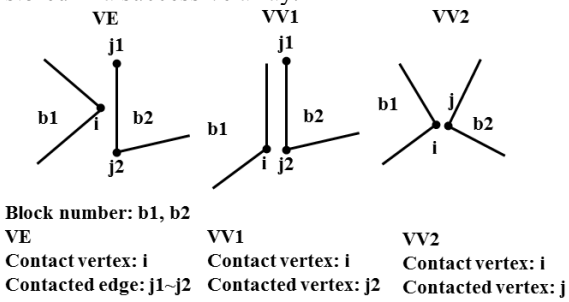

Figure 3. Three typical contact models

Contact transfer is a search process in brief. Each contact of the previous step will search the contacts of the current step. If their contact data are the same, then the contact status parameter, normal displacement, shear displacement, and contact edge ratio of the previous step are transferred to the current step. Sorted search is used to execute the contact transfer on the GPU. First, all the contact data of VE, VV1, or VV2 are combined into a successive array *A*. Second, array *A* is sorted on the basis of the minor block number and then written to array *SA*. Third, the index of each block in array *SA* is computed. Fourth, every half warp (16 threads) gets one contact data of the previous step and searches the contact data of the current step. The matched data are located in a successive interval of array *SA* and can be accessed by the index of the minor block number.

In contact initialization, the contact parameters of VE, VV1, and VV2 are initialized individually. On the basis of the data classification, their workflows are clear and easy to implement on the GPU.

C. Global stiffness matrix building

The difficulty of global stiffness matrix building is located in the non-diagonal part. The global stiffness matrix *A* is naturally blocky and symmetric. Each entry of *A* is a 6×6 sub-matrix. For a blocky system with total block amount *n*, the size of *A* can be treated as $n\times n$. All the diagonal entries are nonzero 6×6 sub-matrixes. Only if block *i* and *j* have contact are their sub-matrix a_{ij} and a_{ji} nonzero. As *A* is symmetric, only the upper entry of *A* is computed and stored.

As mentioned before, the contact data include a contact vertex and a contacted vertex or edge, belonging to different blocks. If the contact vertex belongs to block *i* and the contacted vertex or edge belongs to block *j*, this contact couple will contribute to sub-matrix a_{ii}, a_{jj}, a_{ij}, and a_{ji}. Blocks *i* and *j* usually include several contact data.

Obviously, there exist write conflicts when building matrix *A* on the GPU. Scan and radix sort methods are used to solve this problem when assembling the entries of the same sub-matrix. As depicted in Fig. 4, first, each contact data computes its own sub-matrix in parallel and stores it in array *D*. Second, array *D* is sorted by block number and then written to array *SD*. Third, the boundary position of array *SD* is searched, e.g., *di[i]=(SD[i]-SD[i-1]==0)?1:0*. Fourth, *di* is scanned to produce index *sd1*; then, a similar method to the last step is used to find the end boundary *sd2* of each sub-matrix. Fifth, the sub-matrix index is built using *sd2*; the sub-matrix a_{ii} is the sum from *SD[sd2[i-1]]* to *SD[sd2[i]]*. In practice, all the sort and scan steps act on the block number and index; the data of a sub-matrix are moved only for assembly in the final step. Each step can work on the GPU with high parallelism.

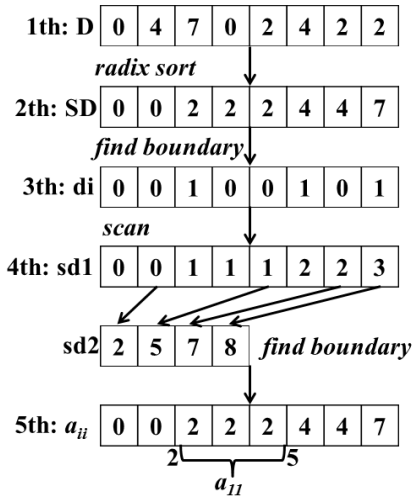

Figure 4. The steps in building the global stiffness matrix on the GPU

D. Interpenetration Checking

The bottleneck of interpenetration checking on the GPU is branch divergence. After data classification branch distribution [34] and restructuring [35] are used to optimize it. For example,

```
if (a == 0)
{
   b = tan(c·d);
   j = fabs(b·e) − fabs(f);
}
if (a == 2)
{
   b = tan(c·d);
   if (e > 0)
   {
      b = 0;
   }
   j = fabs(e)·b − fabs(f)/g;
}
```

This process includes two main branches and one nested branch in the second one. It works well on the CPU, but performs terribly on the GPU owing to branch divergence. To optimize this problem, we can restructure the branches as follows:

```
h = 1;
b = tan(c·d);
if (a == 2)  h = g;
if (a == 0)  b = fabs(b);
if (e·a > 0)  b = 0;
j = fabs(e)·b − fabs(f)/h;
```

All the branches take place only during register writing as the computation has been unified.

IV. SPARSE LINEAR SYMMETRY EQUATION SOLVING

Sparse linear symmetry equation solving is the most time-consuming module of DDA, it usually takes 50% to 90% time in the sequential version. To identify the optimum preconditioner of the CG method for DDA on the GPU, we need to compare several of them. On the other hand, a novel SpMV method for sparse block symmetric matrix is proposed.

A. Preconditioner identification

The preconditioning method transforms the coefficient matrix to a form that has more favorable spectral properties with the same solution. Better spectral properties speed up the convergence rate, and some ill-conditioned matrix can be converged only after the preconditioning.

Fortunately, two facts keep the global stiffness matrix of DDA always in good condition. First, the diagonal sub-matrixes are adjusted by the inertial force in the inner loop. If the simultaneous equations cannot be converged in 200 iterations, then the physical time in the current step should be reduced [1]. The smaller physical time leads the diagonal sub-matrixes to be larger. Second, the equation solution of the previous step is the initial value of the PCG iterative step. The solution of the current step is close to this initial value since the physical time in one step is usually less than 0.0001 s.

Furthermore, the cost of the preconditioner includes the construction and implementation. There is a trade-off between the cost and the gain in the convergence rate. Thus, the preconditioners of DDA on the GPU prefer the low cost in construction and implementation even if their performance is also usually low.

Three different preconditioners were tested to verify this opinion, including BJ, SSOR approximate inverse (SSOR-AI), and ILU. The SSOR-AI preconditioner we used was proposed in [36]. The ILU preconditioner was provided by cuSPARSE [37]. The compute platform and cases are discussed in the Numerical Experience section. The data listed were collected with 1000 time steps. Fig. 5 shows the 26 sampled iterations of the three

preconditioners. The average iterations of the 1000 time steps are listed in Table 1. Generally, the convergence rate of ILU outperformed those of SSOR-AI and BJ by 1.51 and 2.95 times, respectively. However, the construction and implementation cost of ILU were far greater than those of BJ and SSOR-AI. As shown in Fig. 10, the cost of triangular system solving (TSS) was nearly 11 times higher than that of SpMV in cuSPARSE. Even if a level scheduling algorithm was exploited to increase the parallelism in TSS, it only managed to achieve a 20% performance improvement [38]. The final performance can be evaluated by the total time of the equation solving. BJ and SSOR-AI are more advisable for DDA.

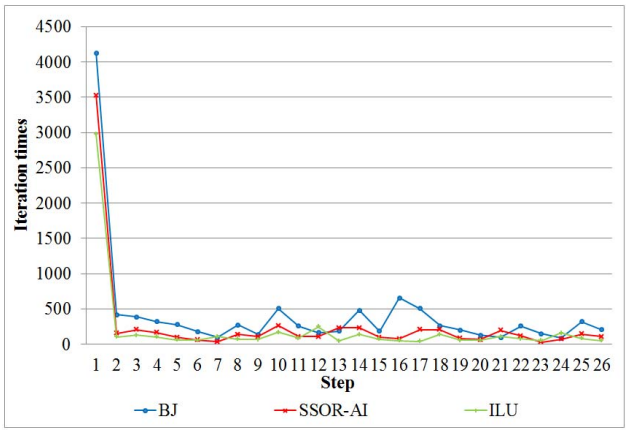

Figure 5. Sampled iterations of the three preconditioners

TABLE I. THE TIME OF THE THREE PRECONDITIONERS

Time	Preconditioners		
	BJ	*SSOR*	*ILU*
Average Iterations/Step	275	141	93
Construction Time (ms)	0.059	0.208	31.465
Implementation Time (ms)	0.011	0.118	7.269
Equation Solving Total Time (ms)	60330	62830	873787

B. Sparse matrix-vector multiplication

The global stiffness matrix of DDA is symmetric. Only the upper triangular matrix parts are computed and stored. Most of the proposed SpMV methods need to recover the global stiffness matrix to a full matrix for accessing GPU global memory in coalescence. However, the cost cannot be ignored in a nested loop in computing since building the global stiffness matrix occurs in every loop. The SpMV method proposed here need not recover the global stiffness matrix and only accesses the upper triangular matrix parts in coalescence once.

We named our storage format for sparse block symmetric matrix as half slice block compressed sparse row format (HSBCSR). Two arrays are used to store nonzero sub-matrix data, namely, *d-data* and *nd-data-up*. Array *d-data* is for diagonal sub-matrixes, whereas *nd-data-up* is for non-diagonal ones of the upper triangular matrix parts. They take the same storage format, which is depicted in Fig. 6. *n* Sub-matrixes (*6×6*) are divided into six slices on the basis of the local row number. The sort priority is slice number, global row number, and then global column number. The length of one slice is a multiple of 32 to satisfy the alignment condition of the GPU's global memory access.

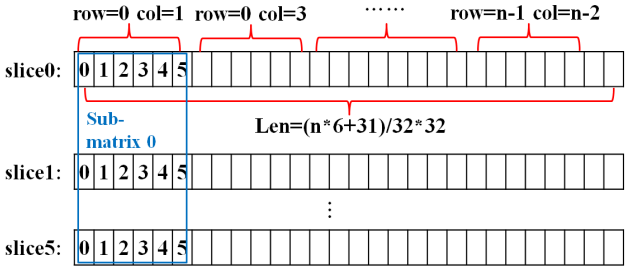

Figure 6. The *d-data*/*nd-data-up* arrays of HSBCSR

Four arrays are used to store the indices of non-diagonal sub-matrixes, namely, *rc, row-up-i, row-low-i*, and *row-low-p*. Array *rc* is a simple compressed storage of the row and column number of each sub-matrix in *nd-data-up*. Array *row-up-i* is the row index of the upper triangular matrix parts, and *row-up-i[i]* is the end position of row *i*.

Assuming array *nd-data-low* is the storage of the non-diagonal sub-matrix of the lower triangular matrix parts, we consider array *row-low-i[i]* to be the end position of row *i*. The global matrix is symmetric, which indicates that the sub-matrix of *nd-data-low* can be transposed to *nd-data-up*. Array *row-low-p* records their mapping relation; *row-low-p[i]=j* means the position of a sub-matrix in *nd-data-low* is *i* and its transposed sub-matrix in *nd-data-up* is *j*. Fig. 7 gives an example to clarify these indices.

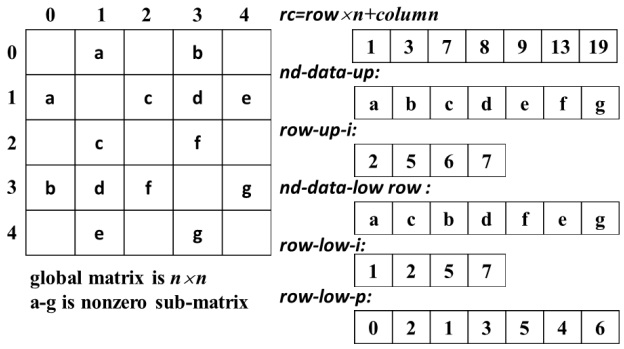

Figure 7. The indices of HSBCSR

The SpMV computation includes two stages. In the first stage, each non-diagonal sub-matrix is multiplied by the related upper vector and lower vector; then, the results are stored in array *up-res* and *low-res*, respectively. The entries of a sub-matrix are accessed slice by slice. The entries of a vector are accessed through texture memory for non-coalesced access. The results of *low-res* among different slices can be added up in the registers directly. The results of *up-res* will be stored in a shared memory first and then reduced together. Fig. 8 illustrates one of these reductions. Each CUDA thread operates with one

row to reduce it for the final result. In the proposed shared memory access scheme, CUDA threads access different banks of the shared memory at the same time. All the entries are reduced concurrently with minimum bank conflicts, and none of CUDA threads will be idle during the process. Moreover, each sub-matrix is treated as an individual in stage 1. Load unbalance issue does not exist in this way.

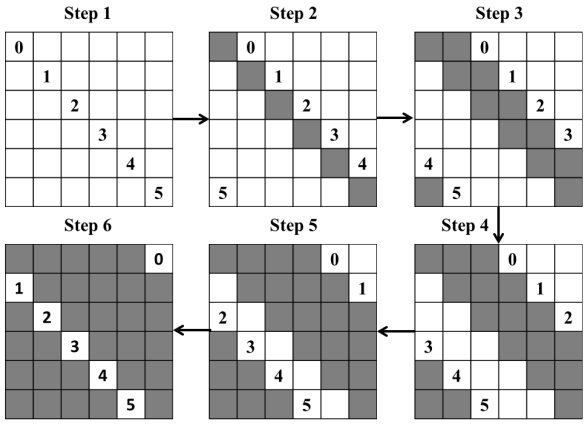

Figure 8. Reducing the results of a sub-matrix by multiplying with the upper vector in the shared memory

The second stage adds up the sub-matrixes multiplied with the vector results belonging to the same row of array *up-res* and *low-res*. Indices *row-up-i, row-low-i*, and *row-low-p* are used here to work it out. The reduction of array *up-res* is regular and fast since the data of every six rows, which are treated as an integer, are located in a successive interval. Every 48 CUDA threads, multiples of 6 and 16, is used to add up the data of six rows from the global memory in coalescence and then reduced in the shared memory.

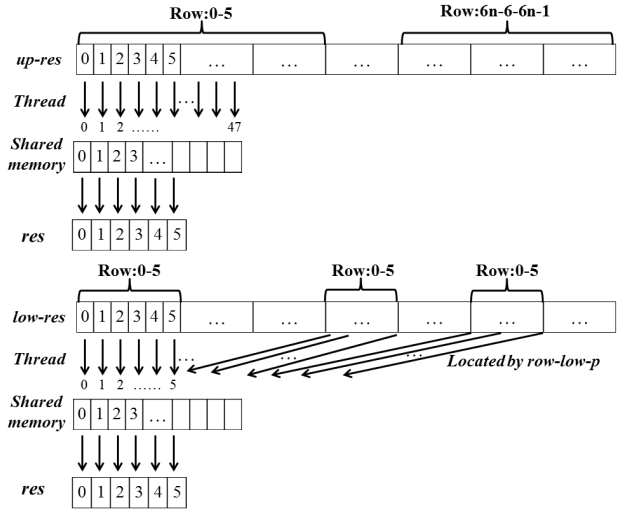

Figure 9. Reduction of array *up-res* and *low-res*

However, the reduction of array *low-res* is more complicated. The data of every six rows are split into a discrete position of array *low-res* irregularly. They cannot be accessed in coalescence from the global memory. Texture memory is adopted to relieve this problem. Every six entries of array *low-res* is still attributed to the successive six rows. Array *row-low-p* is used to locate the data belonging to the same six rows. Fig. 9 depicts the reduction of array *up-res* and *low-res*.

Lastly, the multiplication between diagonal sub-matrixes and vectors is computed and added up to produce the final results. The computation is much more simple and regular by separating diagonal sub-matrixes from non-diagonal ones.

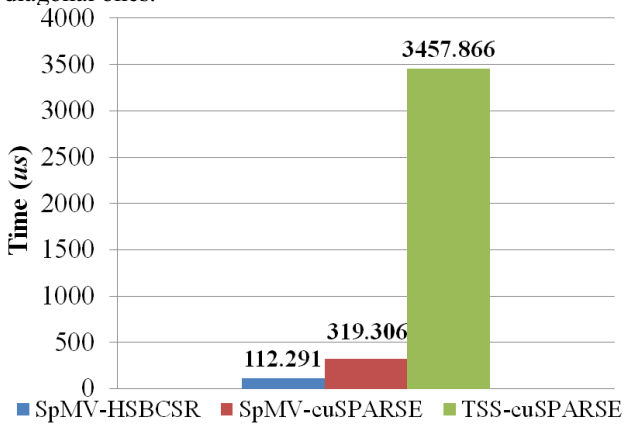

Figure 10. The time of SpMV and TSS on the GPU

The matrix of case 1 in the Numerical Experience section was tested to evaluate the SpMV performance. The number of diagonal sub-matrixes was 4361 and that of the non-diagonal ones was 18731. The time details are depicted in Fig. 10. Our SpMV-HSBCSR method was 2.8 times faster than SpMV-cuSPARSE.

V. NUMERICAL EXPERIENCE

Two typical cases were used to test the performance of GPU-based DDA. Case 1 is a static stability analysis for a realistic slope. Case 2 is a dynamic motion analysis of falling rocks on the slope.

Both of them were tested on a workstation with an Intel Xeon E5620 CPU and an NVIDIA Tesla K20/K40 GPU. Double precision was required in the computation.

A. Case 1

The model of case 1 included 4361 blocks, 5 different block materials, and 38 types of joint materials. It took 40,000 time steps until all the blocks stayed in the static state. The dimension of the global stiffness was (4361×6)×(4361×6). The number of contact detection objects and diagonal sub-matrixes remained 4361; that of non-diagonal ones varied from 2242 to 18731 among different time steps. Fig. 11 shows the initial state, whereas Fig. 12 shows the final static state after the computation.

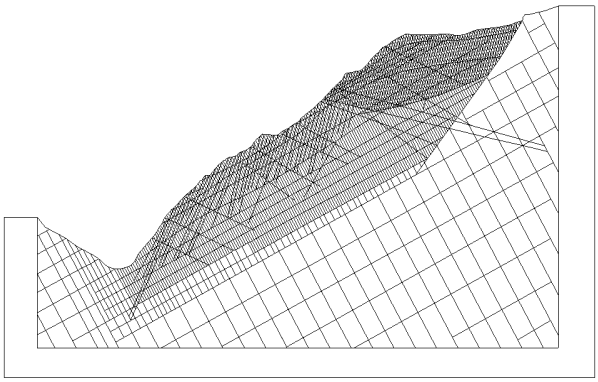

Figure 11. Initial state of case 1

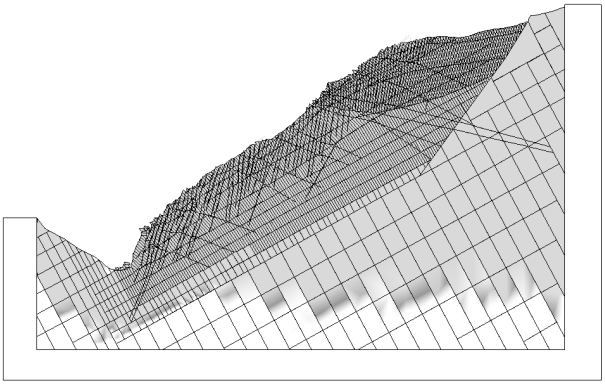

Figure 12. Final static state of case 1

The time costs and speed-up rates of the different modules of case 1 are listed in Table 2. Contact detection reached the highest speed-up rate, which was more than hundreds times. Equation solving took up the bulk of the time and acquired an impressive speed-up rate. The total speed-up rate of case 1 reached 41.94 times on the K20 and 48.72 times on the K40 GPU.

TABLE II. TIME COSTS OF CASE 1

Modules	Case 1 Time (s)			Case 1 Speed-up rate	
	E5620	*K20*	*K40*	*K20*	*K40*
Contact Detection	4975.91	53.4	42.28	93.18	117.69
Diagonal Matrix Building	180.997	2.13	1.68	84.98	107.74
Non-diagonal Matrix Building	1063.25	295.06	242.76	3.6	4.38
Equation Solving	92401.4	1992.1	1723.7	46.38	53.60
Interpenetration Checking	2367.8	63.66	60.04	37.19	39.44
Data Updating	276.081	6.19	5.63	44.6	49.04
Total	101,339	2416.1	2080.2	41.94	48.72

B. Case 2

The purpose of case 2 was to simulate the motion of falling rocks on the slope. The slope was 700 m in height. Initially, 1683 blocks with an average size of 2×2 m^2 stayed at the top of the slope. It took 80000 time steps until all the blocks slid into the bottom of this slope. The scales of case 2 were smaller than those of case 1. However, case 2 took double time steps and consumed more time. Fig. 13 depicts the motion process of case 2.

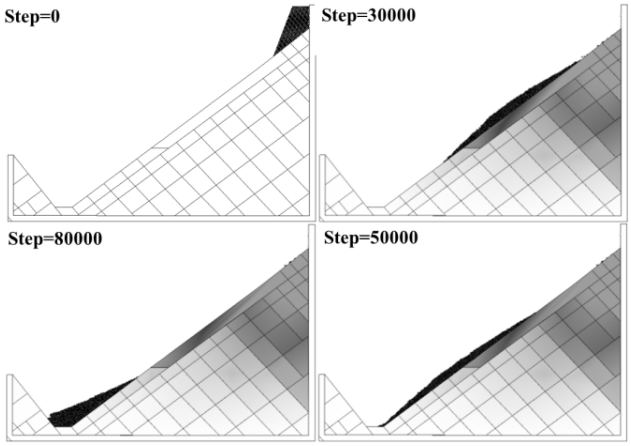

Figure 13. The motion process of case 2

The costs and speed-up rate of the different modules of case 2 are listed in Table 3. The total speed-up rate of case 2 reached only 5.48 times on the K20 and 6.26 times on the K40 GPU. The data size of case 2 was smaller than that of case 1, and the equation solving in the dynamic case was much easier than in the static case. It was related to the way physical time was calculated at each step of DDA.

TABLE III. TIME COSTS OF CASE 2

Modules	Case 2 Time (s)			Case 2 Speed-up rate	
	E5620	*K20*	*K40*	*K20*	*K40*
Contact Detection	5560.61	72.84	59.43	76.34	93.57
Diagonal Matrix Building	122.578	4.78	3.74	25.64	32.77
Non-diagonal Matrix Building	817.912	416.49	343.84	1.96	2.39
Equation Solving	12219.1	3122.7	2755.1	3.91	4.44
Interpenetration Checking	1470.82	96.33	88.73	15.27	16.58
Data Updating	207.091	15.67	13.98	13.22	14.81
Total	20454.9	3731.7	3267.3	5.48	6.26

VI. COUNCLUSION AND FUTURE WORK

In this paper, we presented a complete GPU-based DDA method with a data classification framework. All the compute modules are combined by the scan and radix sort methods to execute on a GPU without data transmission

between the host and the device. A novel SpMV method with a matrix storage format named HSBCSR for a sparse block symmetric matrix was proposed. Moreover, some memory write conflicts and branch divergence issues on the GPU had been solved. The proposed GPU-based DDA outperformed the original CPU-based serial implementation by 48.72 times in the speed-up rates. The speed-up rates were impressive and quite qualified to meet the demands of DDA researchers in terms of efficiency. The next step of this work will focus on applying these efforts to three-dimensional DDA on the multiple GPUs.

ACKNOWLEDGMENT

The authors thank Dr. Genhua Shi for his supervision, guidance, and encouragement throughout this work. This work is supported by the National Natural Science Foundation of China (nos. 61471338 and 61303155), the Knowledge Innovation Program of the Chinese Academy of Sciences, the President Fund of UCAS, the CRSRI Open Research Program (CKWV2015217/KY), and the Beijing Municipal Natural Science Foundation (grant no. 4131004).

REFERENCES

[1] G. H. Shi, Discontinuous deformation analysis: A new numerical model for the statics and dynamics of block systems, University of California, Berkeley, 1988

[2] "Discontinuous Deformation analysis," https://en.wikipedia.org/wiki/Discontinuous_Deformation_Analysis#cite_note-Chang96-4.

[3] Y. Ning, Z. Yang, B. Wei, and B. Gu, "Advances in Two-Dimensional Discontinuous Deformation Analysis for Rock-Mass Dynamics," International Journal of Geomechanics, 10.1061/(ASCE)GM.1943-5622.0000654 , E6016001, 2016.

[4] S. Cook, CUDA programming: A developer's guide to parallel computing with GPUs, Morgan Kaufmann, San Francisco, 2012.

[5] Y. Shigeto, M. Sakai, S. Koshizuka and Y. Yamada, "GPU Accelerated Simulation for Discrete Element Method," Journal of the Research Association of Powder Technology Japan, 2008, 45(11):758-765.

[6] L. Wang, S. Li, G. Zhang, Z. Ma, and L. Zhang, "A GPU-based parallel procedure for nonlinear analysis of complex structures using a coupled FEM/DEM approach," Mathematical Problems in Engineering, ID 618980, 15pages, 2013.

[7] Z. Fu, T. J. Lewis, R. M. Kirby, and R. T. Whitaker, "Architecting the finite element method pipeline for the GPU," Journal of computational and applied mathematics, vol. 257, 2014, pp.195-211.

[8] Q. Miao, M. Huang and Q. Wei, "Parallel computing of numerical manifold method with OpenMP," Information, Computing and Telecommunication, YC-ICT'09. IEEE Youth Conference on. IEEE, pp. 174-177, 2009.

[9] X. Fu, Q. Sheng and Y. Zhang, "Parallel computing method for discontinuous deformation analysis using OpenMP," Rock and Soil Mechanics, 35(8): 2401-2407, 2014.

[10] Y. Xiao, Q. Miao, M. Huang, Y. Wang, and J. Xue, "Parallel Computing of Discontinuous Deformation Analysis Based on Graphics Processing Unit," International Journal of Geomechanics, 10.1061/(ASCE)GM.1943-5622.0000717 , E4016010,2016.

[11] H. Nguyen, Gpu gems , Addison-Wesley Professional, 2007.

[12] C. Lauterbach, Q. Mo, D. Manocha, "gProximity: hierarchical GPU•based operations for collision and distance queries," Computer Graphics Forum. Blackwell Publishing Ltd, 2010, 29(2): 419-428.

[13] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," The International Journal of Robotics Research, 2011: 0278364911429335.

[14] T. H. Wong, G. Leach and F. Zambetta, "An adaptive octree grid for GPU-based collision detection of deformable objects," The Visual Computer, 30(6-8): 729-738, 2014.

[15] N. Govender, D. N. Wilke, S. Kok, and R. Els, "Development of a convex polyhedral discrete element simulation framework for NVIDIA Kepler based GPUs," Journal of Computational and Applied Mathematics, 270: 386-400, 2014.

[16] F. Geleri, O. Tosun and H. Topcuoglu, "Parallelizing Broad Phase Collision Detection Algorithms for Sampling Based Path Planners," Parallel, Distributed and Network-Based Processing (PDP), 21st Euromicro International Conference on. IEEE, 2013: 384-391.

[17] R. Li, and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," The Journal of Supercomputing, 63(2): 443-466, 2013.

[18] S. Georgescu, P. Chow and H. Okuda, "GPU acceleration for FEM-based structural analysis," Archives of Computational Methods in Engineering, 20(2): 111-121, 2013.

[19] X. Wang, W. Xue, J. Zhai, Y. Xu, W. Zheng, and Lin, H. "A fast tridiagonal solver for Intel MIC architecture." Parallel and Distributed Processing Symposium, 2016 IEEE International. 2016: 172-181.

[20] M. Wang, H. Klie, M. Parashar and H. Sudan, "Solving sparse linear systems on NVIDIA Tesla GPUs," Computational Science–ICCS 2009. Springer Berlin Heidelberg, pp. 864-873, 2009.

[21] N.Bell, S. Dalton and L.N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," SIAM Journal on Scientific Computing, 34(4): C123-C152, 2012.

[22] R. Gandham, K. Esler, and Y. Zhang "A GPU accelerated aggregation algebraic multigrid method," Computers & Mathematics with Applications, 68(10): 1151-1160, 2014.

[23] M. M. Dehnavi, D. M. Fernandez, J. L. Gaudiot, and D. D. Giannacopoulos, "Parallel sparse approximate inverse preconditioning on graphic processing units," Parallel and Distributed Systems, IEEE Transactions on, 24(9): 1852-1862, 2013.

[24] N.Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[25] F. Vazquez, G. Ortega, J. J. Fernández and E. M. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," Computer and Information Technology (CIT), IEEE 10th International Conference, 2010, pp.1146-1151.

[26] A. Dziekonski, A. Lamecki and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a GPU," Progress in Electromagnetics Research, 2011,116: 49-63.

[27] J. Wong, E. Kuhl and E. Darve, "A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems," International Journal for Numerical Methods in Engineering, 102(12): 1784-1814, 2015.

[28] M. Verschoor and A. C. Jalba, "Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs," Parallel Computing, 2012, 38(10): 552-575.

[29] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland, "Optimizing Sparse Matrix Operations on GPUs Using Merge Path," Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. 2015: 407-416.

[30] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14, 2009.

[31] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," Parallel Processing Letters, 21(02): 245-272, 2011.

[32] "Faster Parallel Reductions on Kepler," https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[33] "NVIDIA Nsight", http://www.nvidia.com/object/nsight.html.

[34] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU," Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011.
[35] I. Chakroun, M. Mezmaz, N. Melab and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm," Concurrency and Computation: Practice and Experience, 25(8): 1121-1136, 2013.
[36] H. Rudi and J. Koko. "Parallel preconditioned conjugate gradient algorithm on GPU," Journal of Computational and Applied Mathematics, 236, 15: 3584-3590, 2012.
[37] "cuSPARSE."http://docs.nvidia.com/cuda/cusparse.
[38] Y. Chen, Y. Zhao, W. Zhao. "A Comparative Study of Preconditioners for GPU-accelerated Conjugate Gradient Solver," 15th IEEE International Conference on High Performance Computing and Communications & 11th IEEE International Conference on Embedded and Ubiquitous Computing. 2014. P628-635