Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 618980, 15 pages
http://dx.doi.org/10.1155/2013/618980

Research Article

A GPU-Based Parallel Procedure for Nonlinear Analysis of Complex Structures Using a Coupled FEM/DEM Approach

Lixiang Wang,[1] Shihai Li,[1] Guoxin Zhang,[2] Zhaosong Ma,[1] and Lei Zhang[2]

[1] *Institute of Mechanics, Chinese Academy of Sciences, Beijing 100190, China*
[2] *State Key Laboratory of Simulation and Regulation of Water Cycle in River Basin, China Institute of Water Resources and Hydropower Research, Beijing 100038, China*

Correspondence should be addressed to Shihai Li; shli@imech.ac.cn

Received 18 July 2013; Accepted 9 September 2013

Academic Editor: Zhiqiang Hu

This study reports the GPU parallelization of complex three-dimensional software for nonlinear analysis of concrete structures. It focuses on coupled thermomechanical analysis of complex structures. A coupled FEM/DEM approach (CDEM) is given from a fundamental theoretical viewpoint. As the modeling of a large structure by means of FEM/DEM may lead to prohibitive computation times, a parallelization strategy is required. With the substantial development of computer science, a GPU-based parallel procedure is implemented. A comparative study between the GPU and CPU computation results is presented, and the runtimes and speedups are analyzed. The results show that dramatic performance improvements are gained from GPU parallelization.

1. Introduction

Many authors have studied the resolution of the nonlinear concrete structure problems, focusing rigorously on the three-dimensional (3D) elastoplastic problem, such as Roca and Marí [1] and Spacone et al. [2]. These authors used a finite element model. Other authors (e.g., Rousseau et al. [3], Villard et al. [4]) proposed a coupled FEM/DEM approach with regard to nonlinear material analysis of structures. As is shown, the modeling of a large structure by means of FEM or DEM may lead to prohibitive computation times. Thus, some authors (e.g., Sziveri et al. [5], Romero et al. [6]) developed CPU-based parallel procedures. With significant development of computer science, the implementation of a GPU-based parallel procedure becomes possible.

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to display [7]. GPU has become an integral part of today's mainstream computing systems. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart [8]. Over the past decade, there has been a marked increase in the performance and capabilities of GPUs. As a result, computing on a GPU card has been a hot topic for scientific research in different numerical areas [9–21]. The techniques, which enable GPUs to efficiently undertake large-scale scientific computing tasks, can be summarized as follows [21].

(i) A GPU card consists of a number of multiprocessors (as shown in Figure 1). On each processor, there are several hundred coresident threads that can execute integer, single and double precision calculations simultaneously.

(ii) Fine-grained parallelization with a very large number of threads is the kernel principal of GPU computing.

(iii) The memory access technique is specially designed in a GPU to accelerate the memory access speed of the threads. To achieve near peak memory access performance, memory coalescing is considered in the design of the data structure and the computational arrangement for computations.

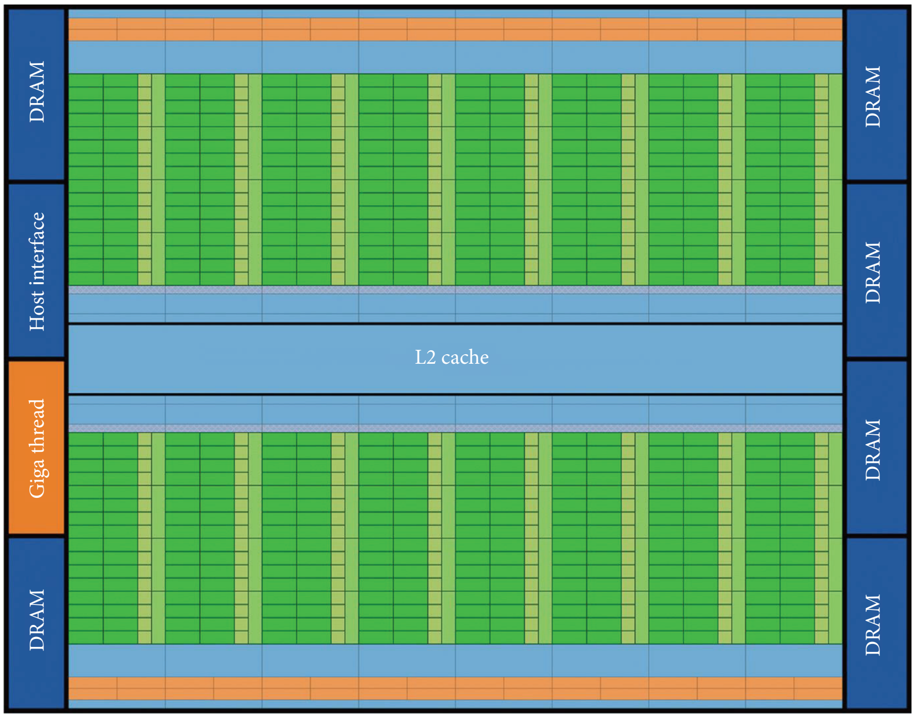


Figure 1: Fermi architecture—Fermi's 16 SMs are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contains an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache) [23].

(iv) The release of Compute Unified Device Architecture (CUDA) makes it much more straightforward to implement GPU code.

CUDA is a parallel computing platform and programming model (as shown in Figure 2) created by NVIDIA and implemented by the GPUs that they produce [22]. It enables dramatic increases in computing performance by harnessing the power of the GPU. It is the most popular programming toolkit in computational sciences. It has already been used by researchers to parallelize their codes of different numerical methods, for example, Molecular Dynamic (MD) [9, 10], Lattice Boltzmann Method (LBM) [11, 12], Boundary Element Method (BEM) [13], Finite Element Method (FEM) [14–16], Discrete Element Method (DEM) [17, 18], Finite Difference Method [19], Moving Particle Semi-implicit (MPS) method [20], and Distinct Lattice Spring Model (DLSM) [21]. The overall speedups of GPU parallel codes compared to serial CPU counterparts are reported to be from 10x to 100x [9–21]. This is promising for both scientific computations and engineering practices.

To model the nonlinear response of complex structures, we will parallelize the coupled FEM/DEM code on a GPU with CUDA. In the first section, the coupled FEM/DEM approach with its basic formulations is introduced. Then, the GPU implementation and optimization techniques are presented. Furthermore, the parallel FEM/DEM code is tested on computers equipped with modern GPU cards. Finally, conclusions regarding the GPU parallelization are drawn.

2. Coupled Finite/Discrete Element Model

2.1. Geometric Representation. Li et al. [24, 25] proposed a Continuum-based Discrete Element Method (CDEM) which in fact is a Coupled Finite/Discrete Element Method and has a variety of applications. A detailed description about CDEM will be introduced in the following context.

CDEM is a combination of Finite Element Method (FEM) and Discrete Element Method (DEM). Figure 3 shows the geometric domains used by the CDEM approach. It contains two kinds of elements, blocks (i.e., A, B, C, and D) and contacts (Figures 3(c) and 3(d)). A discrete block consists of one or more FEM elements, all of which share the same nodes and faces. In this paper, it is assumed that a discrete block consists of only one FEM element. All three element types are shown in Figure 4. A contact contains several normal and tangent springs (Figure 5); each connects nodes of neighboring blocks. Inside the block, the FEM is used, while for the interface, the DEM is adopted.

2.2. Governing Equations. Block stress analysis means figuring out stress of every single block and the interaction between different blocks. For every block in the model, it needs to satisfy the following governing equations.

Equilibrium equation:

$$\sigma_{ij,j} + f_i - \rho\ddot{u}_i - \alpha\dot{u}_i = 0. \qquad (1)$$

Strain-displacement relationship:

$$\varepsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right). \qquad (2)$$

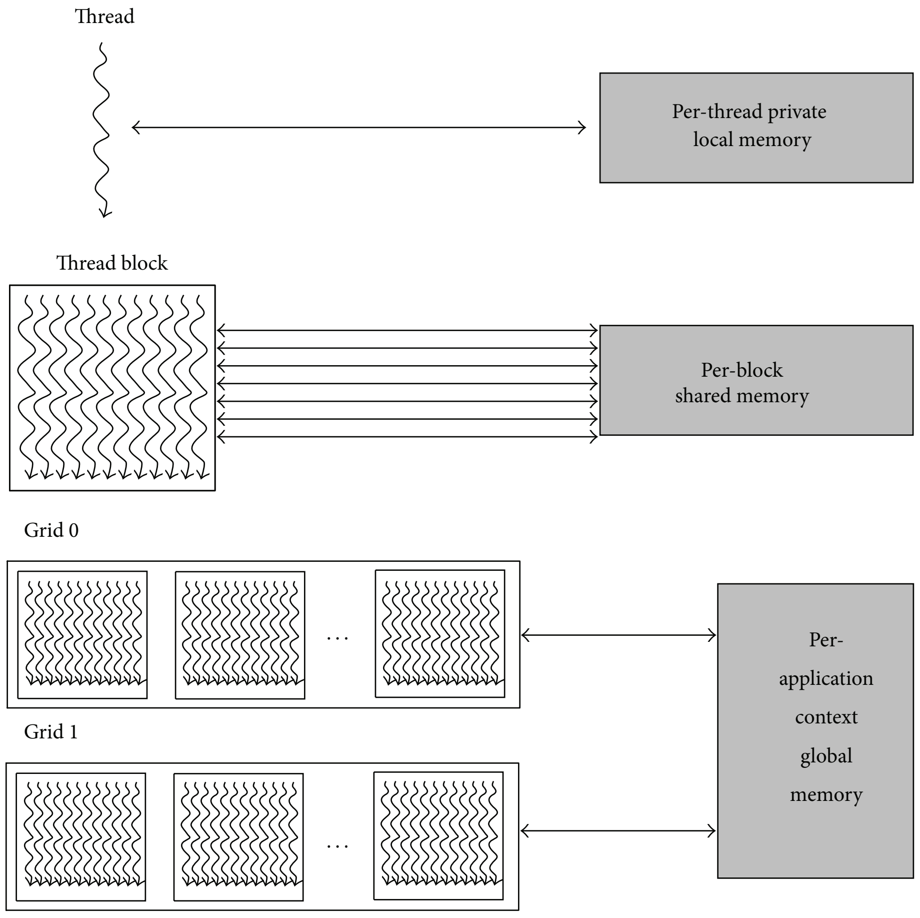

Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application context global memory

Figure 2: CUDA programming model—hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared and per-application global memory spaces [23].

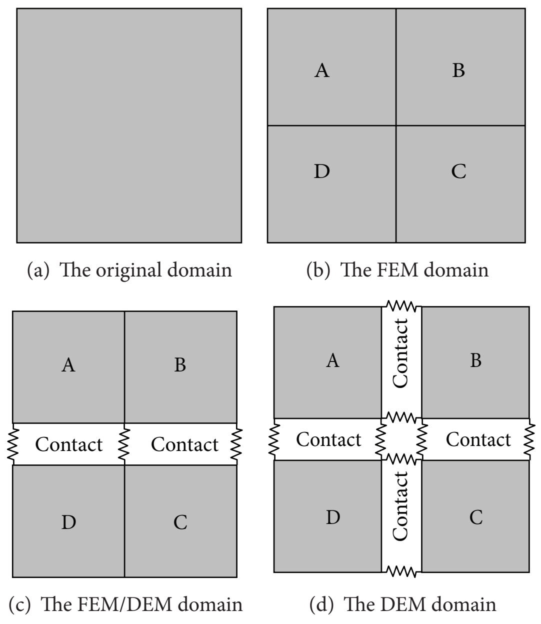


Figure 3: Geometric domains used by CDEM.

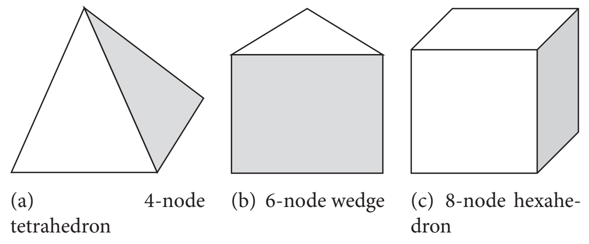


Figure 4: Finite element types adopted in CDEM.

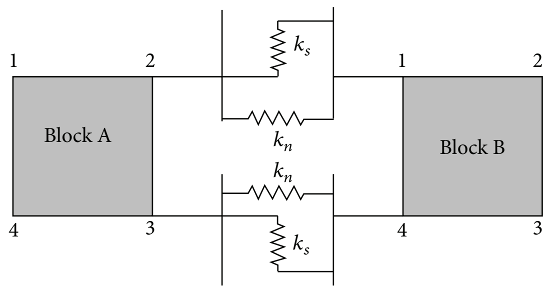


Figure 5: CDEM contact—A 2D contact contains 1 normal spring (k_n) and 1 tangent spring (k_s). The figure illustrates contact A2-B1 and contact A3-B4.

Constitutive law:

$$\sigma_{ij} = \mathbf{D}_{ijkl}\varepsilon_{kl}. \tag{3}$$

Boundary conditions:

$$\begin{aligned} u_i &= \bar{u}_i, \\ \sigma_{ij} n_j &= \bar{t}_i. \end{aligned} \tag{4}$$

Initial conditions:

$$\begin{aligned} u_i(x,y,z,0) &= u_i^0(x,y,z), \\ u_{i,t}(x,y,z,0) &= u_{i,t}^0(x,y,z). \end{aligned} \tag{5}$$

In (1)~(5), $\sigma_{ij,j}$ represents the first-order partial derivative of stress tensor versus coordinate; f_i stands for the body force; ρ is density; $\ddot{u}_i$ denotes the acceleration; $\dot{u}_i$ denotes the velocity; α is damping ratio; $u_{i,j}$, $u_{j,i}$ are both the first-order partial derivatives of displacement versus coordinate; ε_{ij}, ε_{kl} are strains; σ_{ij} is stress; $\mathbf{D}_{ijkl}$ is stress-strain tensor.

It should be mentioned that various constitutive models in (3) can be used in the block, including the linear elastic model, Drucker-Prager model, the block breakage model, and the discrete spring model.

2.3. Thermal Analysis. The equilibrium equation of heat transfer can be written as

$$\lambda\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right)-c\rho\frac{\partial T}{\partial \tau}=0. \tag{6}$$

The initial condition is

$$T = T_0(x,y,z), \qquad (\tau = 0). \tag{7}$$

The boundary conditions are

$$\begin{aligned} T &= T_b \quad \text{Dirichlet B.C.}, \\ -\lambda\left(\frac{\partial T}{\partial n}\right) &= q \quad \text{Neumann B.C.}, \\ -\lambda\left(\frac{\partial T}{\partial n}\right) &= \beta\left(T - T_a\right) \quad \text{Mixed B.C.} \end{aligned} \tag{8}$$

In (6)~(8), τ represents age; λ is thermal conductivity; ρ stands for density; c is specific heat capacity; T is temperature; T_0 is the initial temperature of concrete; T_b is the fixed boundary temperature; q represents heat flux; T_a denotes ambient temperature in natural convection conditions and adiabatic temperature of boundary layer in forced convection conditions; β is heat transfer coefficient in surface.

By using the variation formulation, the equilibrium equation of heat transfer in (7) can be transformed into the following matrix form:

$$[C]\left\{\dot{T}\right\} + [K]\{T\} = \{Q\}, \tag{9}$$

where $[C]$ is the matrix for specific heat capacity, $[K]$ is the matrix for heat conductivity, and $\{Q\}$ is the heat flux vector.

A time forward difference scheme is used for thermal analysis. Thus, the time difference equation can be formulated as

$$[C]\frac{\{T_{n+1}\}-\{T_n\}}{\Delta\tau} + [K]\{T_n\} = \{Q\}, \tag{10}$$

where $\{T_n\}$ stands for the temperature vector at time step n; $\Delta\tau$ is the time step.

2.4. Dynamic Relaxation Method. By using the variation formulation, the equilibrium equation of momentum in (1) can be transformed into the following matrix form in an element:

$$[M]\{\ddot{u}\} + [C]\{\dot{u}\} + [K]\{u\} = \{G\}, \tag{11}$$

where $[M]$ represents the diagonal mass matrix; $[C]$ represents the damping matrix; $[K]$ represents the stiffness matrix; $\{u\}$ represents the vector of displacement; $\{G\}$ represents the vector of external force.

In our approach, global stiffness matrix is not assembled. Instead, (11) is iterated element by element, using the dynamic relaxation method. In every element, its displacement satisfies (11) strictly. If the solution of every element satisfies (11), then the overall solution made up of element solutions must satisfy the assembled equation. In fact, this is beneficial for the parallel implementation.

In time domain, an explicit iteration technique is applied. In this technique, the acceleration is iterated by the central

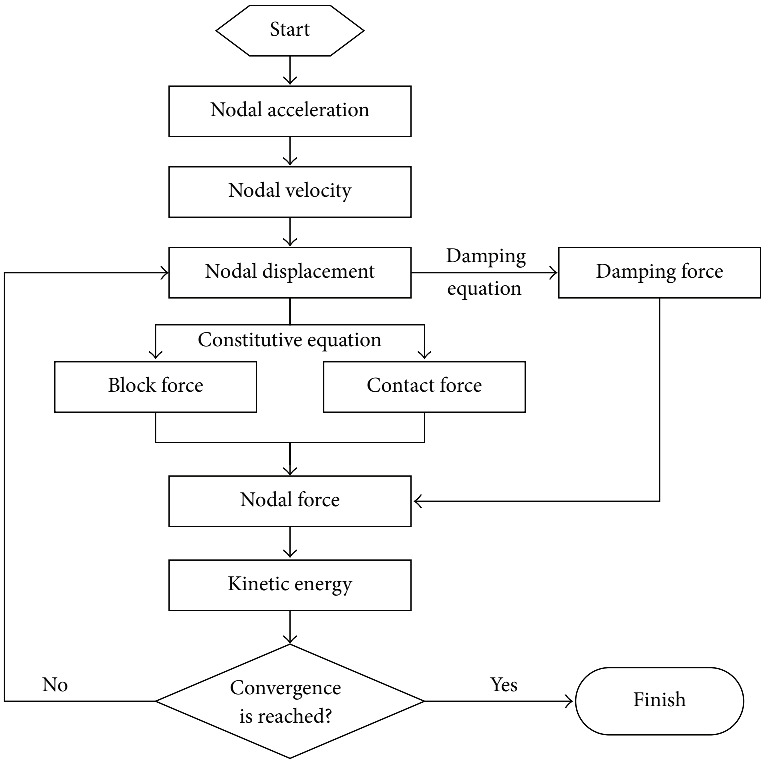


Figure 6: Flow chart of iteration using dynamic relaxation method.

difference scheme, while the velocity is iterated by the unilateral difference scheme. The schemes can be written as

$$\begin{aligned}\ddot{u}_{li}^{n} &= \frac{u_{li}^{n+1} - 2u_{li}^{n} + u_{li}^{n-1}}{(\Delta t)^2} \\ &= \frac{\left(u_{li}^{n+1} - u_{li}^{n}\right)/\Delta t - \left(u_{li}^{n} - u_{li}^{n-1}\right)/\Delta t}{\Delta t}, \\ \dot{u}_{li}^{n+1} &= \frac{u_{li}^{n+1} - u_{li}^{n}}{\Delta t},\end{aligned} \tag{12}$$

where $\ddot{u}_{li}^{n}$, $\dot{u}_{li}^{n}$, and u_{li}^{n}, respectively, represent the acceleration, the velocity, and the displacement of the ith node of the lth element, at the nth time step.

The explicit iteration technique can be formulated from (12):

$$\begin{aligned}\dot{u}_{li}^{n+1} &= \dot{u}_{li}^{n} + \ddot{u}_{li}^{n}\Delta t, \\ u_{li}^{n+1} &= u_{li}^{n} + \dot{u}_{li}^{n+1}\Delta t.\end{aligned} \tag{13}$$

The process of explicit iteration using the dynamic relaxation method is illustrated in Figure 6. During the iteration, convergence is reached when the total magnitude of the kinetic energy is minimized.

3. GPU Implementation

3.1. GPU Implementation with CUDA. In Section 2, we refer to the CDEM approach, in which global stiffness matrix is not assembled. An element-by-element strategy is employed to solve the equilibrium equation. In this sense, the CDEM is suitable to run on a GPU since calculations involved in the CDEM are all performed independently for the elements and the nodes.

In GPU parallelization, heterogeneous computing methodology is usually used. This means that the serial part of the code executes in a *host* (CPU) thread, while the parallel part executes in a large amount of *device* (GPU) threads. The *device* can be viewed as a virtual computer that has its own separate memory space. It is ideally suited to perform element and nodal calculations across hundreds of threads. Figure 7 shows a flow chart of the GPU implementation of the CDEM. One principle for the implementation is to replace the original CPU-based calculation functions with CUDA kernels. A kernel is a function that runs on the *device*. Another principle is to integrate as many CUDA kernels into a large one as possible. This is to minimize the delay of kernel launching. The kernels should be integrated only when they can be parallelized. Serial and dependent kernels cannot be integrated together.

An example is given to demonstrate the GPU implementation with CUDA. In the example, the CPU functions *DoElems* and *DoNodes* are first replaced by corresponding CUDA kernels. Then, the CUDA kernels are integrated into one CUDA kernel *DoKernel*. The additional qualifier "__global__" is used to define a CUDA kernel. Embellished with <<<*nBlocks, nThreads*>>>, the *DoKernel* function will be run in *nBlocks* × *nThreads* threads in parallel (see Algorithms 1 and 2).

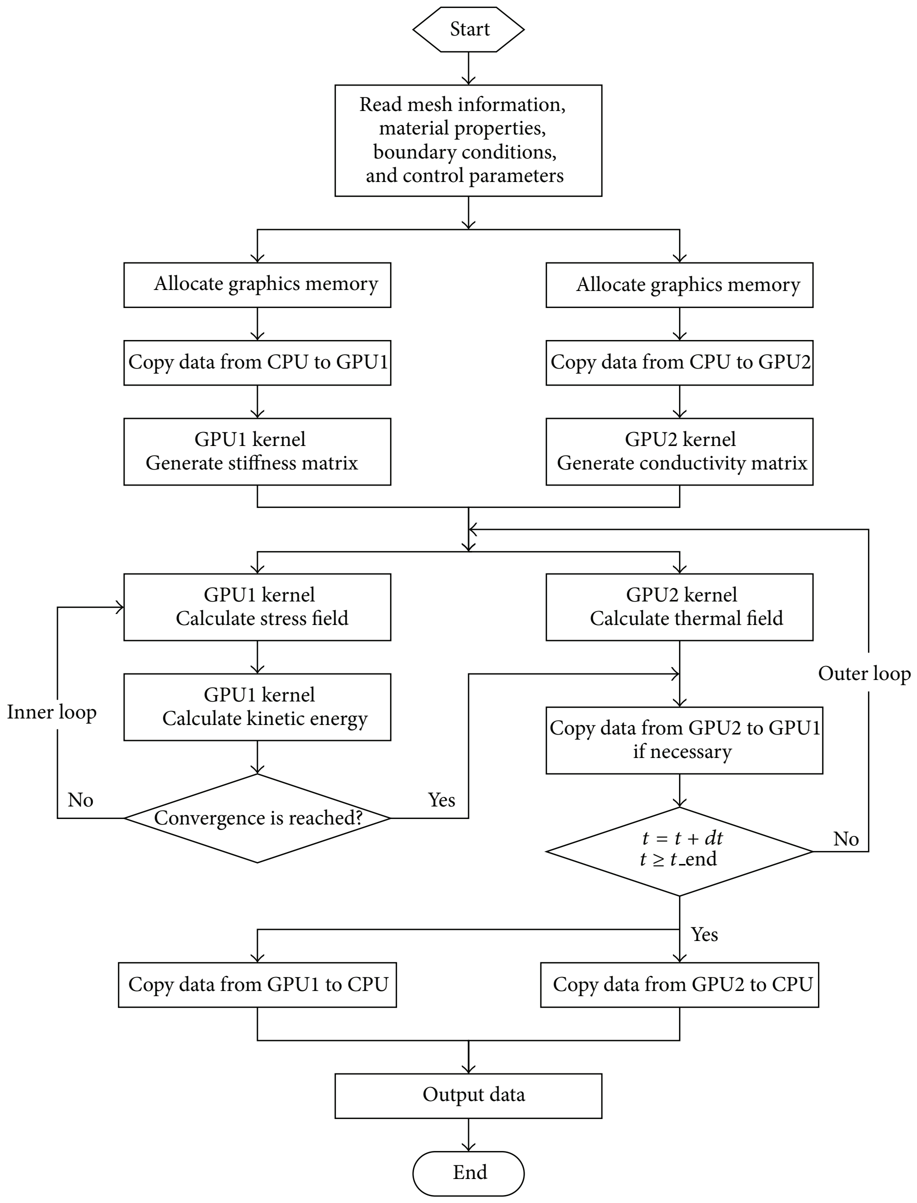

Start
Read mesh information, material properties, boundary conditions, and control parameters
Allocate graphics memory
Allocate graphics memory
Copy data from CPU to GPU1
Copy data from CPU to GPU2
GPU1 kernel Generate stiffness matrix
GPU2 kernel Generate conductivity matrix
GPU1 kernel Calculate stress field
GPU2 kernel Calculate thermal field
Outer loop
GPU1 kernel Calculate kinetic energy
Inner loop
Copy data from GPU2 to GPU1 if necessary
No
Convergence is reached?
Yes
t = t + dt
t ≥ t_end
No
Yes
Copy data from GPU1 to CPU
Copy data from GPU2 to CPU
Output data
End

Figure 7: Flow chart of GPU implementation.

As illustrated in Figure 7, stress and thermal fields are analyzed on two GPUs concurrently. Such multi-GPU technique ensures higher efficiency when we perform multifield simulation for complex structures. All of the data in two fields will be first sent to the *device*. During the calculation, only thermal stress will be read from one GPU and sent to another. The communication is realized by using the CUDA function *cudaMemcpy*. The data transfer from the *device* to the *host* only happens when the simulation results need to be output. Since the data are kept on the *device* for the whole simulation process, memory transfer between *host* and *device* is largely reduced. Data transfer from GPU2 to GPU1 only happens under necessary circumstances, which will not affect the overall efficiency.

3.2. GPU Parallel Algorithm and Procedure. The GPU parallel algorithm is realized by GPU kernels. In Section 3.1, we have discussed the implementation of one GPU kernel. Now all the kernels will be presented to demonstrate the detailed process (see Algorithm 3).

The kernels *GenerateStiffness()* and *GenerateConductivity()* generate element stiffness matrix and element conductivity matrix, respectively. They use similar algorithms. To generate element matrix, we perform the Gaussian integrations over each element by using the following formula:

$$[K]^e = \int_{\Omega^e} [B]^T [D] [B] \, dx \, dy \, dz$$

```
// CPU functions
void DoElems ( /*parameters omitted*/ )
{
// omitted codes that do element calculation;
}
void DoNodes ( /*parameters omitted*/ )
{
// omitted codes that do nodal calculation;
}
void main ( )
{
        // Element and nodal calculation
        DoElems( /*parameters omitted*/ );
        DoNodes( /*parameters omitted*/ );
}
```

Algorithm 1

```
// GPU kernels
__global__ void DoKernel
( /*parameters omitted*/ )
{
// omitted codes that do element calculation;
// omitted codes that do nodal calculation;
}

void main ( )
{
        // Element and nodal calculation
        DoKernel <<< nBlocks, nThreads >>>
        ( /*parameters omitted*/ );
}
```

Algorithm 2

$$= \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} [B]^T [D] [B] |J| \, d\xi \, d\eta \, d\zeta$$
$$= \sum_{i,j,k=1}^{n} w_i w_j w_k [B]_G^T [D] [B]_G |J| , \tag{14}$$

where $[B]$ is strain matrix; $[D]$ is stress-strain tensor; $|J|$ is the determinant of Jacobian matrix; w_i, w_j, and w_k are weights in Gaussian integrations.

The strain matrix $[B]$ is calculated by GPU. For linear problems, $[B]$ is calculated and stored in GPU memory before iteration. For memory-consuming problems (i.e., nonlinear or big problems), $[B]$ is calculated when iteration is taking place. Since the GPU calculation of $[B]$ matrix is fast, it costs little time. For different elements, the parallel algorithms to generate $[B]$ matrix are different. The algorithm is shown below.

Algorithm of Generation of $[B]$ *Matrix*. (1) For each 4-node tetrahedron element, there is one or four Gaussian integration points. Each element is divided into four threads. If there is one Gaussian integration point, the first thread performs the Gaussian integration and the weight is unit. If there are four Gaussian integration points, each of the four threads performs its own Gaussian integration and each of the four weights is 0.25.

(2) For each 6-node wedge element, there are six Gaussian integration points. Every 8 threads are one group. In one group, only the first six threads perform the Gaussian integrations, while the left two threads do nothing.

(3) For each 8-node hexahedron element, there are eight Gaussian integration points. Each element is divided into eight threads. Each thread performs the Gaussian integration of one Gaussian point.

The kernel *DoKernel()* does nodal and element calculation of stress field. The algorithm in *DoKernel()* can be summarized as follows.

Algorithm of DoKernel(). (1) Calculate external nodal forces $\{F\}^{\text{ext}}$, which include tractions and body forces.

(2) Calculate internal nodal forces: $\{F\}^{\text{int}} = [K]^e\{u\}^e + [C]^e\{\dot{u}\}^e$ (in static problems, $\{F\}^{\text{int}} = [K]^e\{u\}^e$).

(3) Calculate total nodal forces: $\{F\}^{\text{tot}} = \{F\}^{\text{ext}} - \{F\}^{\text{int}}$.

(4) Calculate nodal accelerations by Newton's Second Law: $\{a\} = [M]^{-1}\{F\}^{\text{tot}}$.

(5) Calculate nodal velocities $\{v\}$ and displacements $\{u\}$ by (13).

(6) Repeat 1~5 until convergence is reached.

Steps 1~3 are calculations in terms of elements, while steps 4~5 are in terms of nodes. In parallel algorithm, all elements and nodes are distributed into *nBlocks* × *nThreads* threads. Thus, the *DoKernel()* function will be run in *nBlocks* × *nThreads* threads in parallel.

It is worth mentioning that a CUDA library function *__syncthreads()* must be called between Steps 1 and 2. The matrix multiplication $[K]^e\{u\}^e$ is performed by GPU. It demands that the displacement vector $\{u\}$ be synchronized. Thus, the function *__syncthreads()* is used to keep all the displacement components at the same level.

Boundary conditions need special treatments. For $u = 0$ boundary, no calculation is done. The displacements are set as zero. For traction boundary, tractions are equivalent to nodal forces. Each thread performs the calculation of one node.

Convergence is reached if the kinetic energy is lower than a threshold. The kinetic energy is defined as.

$$E_k = \sum \frac{1}{2}\{v\}^T \{v\} . \tag{15}$$

3.3. Performance Optimization. The parallel procedure is implemented and tested by using NVIDIA CUDA Toolkit 3.0. The 8-node hexahedron element is adopted. The 6-node wedge element can be viewed as a degenerated element from 8-node hexahedron. The 4-node tetrahedron element owns half the number of nodes, compared with the 8-node hexahedron element. They share a proportional (1/2) relationship with each other. Thus, a proportional optimization strategy can be used. In this sense, once the 8-node hexahedron element is optimized, the 4-node tetrahedron element is optimized as well by multiplying a proportion of 1/2.

```
// GPU kernels for stress analysis
__global__ void GenerateStiffness();        // To generate element stiffness matrices
__global__ void DoKernel();                 // To do nodal and element calculation of stress field
__global__ void CalcEnergy();               // To calculate kinetic energy
// GPU kernels for thermal analysis
__global__ void GenerateConductivity();     // To generate element conductivity matrices
__global__ void DoKernelT();                // To do nodal and element calculation of thermal field
```

Algorithm 3

Table 1: Specifications of the GPUs [26–28] used in this study.

Model	GeForce GTX580	GeForce GTX670	GeForce GTX680
GPU	GF110	GK104	GK110
Number of CUDA cores	512	1344	1536
Number of SMs	16	8	8
Processor clock (MHz)	1544	980	1006–1058
Memory	3 GB DDR5	4 GB DDR5	2 GB DDR5
Memory bandwidth (GB/s)	192.4	192.2	192.2

To optimize the parallel procedure, the number of threads per block needs to be carefully chosen. As is recommended by NVIDIA [29], the best results will be achieved when the number of threads per block is a multiple of 64. For ideal performance, this number is recommended to be over 192. In practice, the best performance is achieved when the number of elements per block is 16 or 32. Equations (16) [18] and (17) show the relationship between the number of threads per block N_t and the number of elements per block N_e.

8-node/6-node elements:

$$N_t = N_e \cdot 8 = \begin{cases} 256, & N_e = 32, \\ 128, & N_e = 16. \end{cases} \quad (16)$$

4-node element:

$$N_t = N_e \cdot 4 = \begin{cases} 128, & N_e = 32, \\ 64, & N_e = 16. \end{cases} \quad (17)$$

The effective bandwidth of each memory space depends significantly on the memory access pattern. As described in [29], global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32, 64, or 128 bytes. To achieve maximum global memory bandwidth, the data structures are realigned to thread. Besides, shared memory is used to store nodal variables, for the shared memory space is much faster than the local and global memory spaces, and nodal variables are reused among threads [18].

4. Performance Tests

4.1. Test Platform. In all our tests, three computers, one Intel Xeon E5506 at 2.13 GHz with NVIDIA GeForce GTX580, one Intel Xeon E5-26090 at 2.40 GHz with NVIDIA GeForce GTX670, and one Intel Core i5 2320 at 3.20 GHz with NVIDIA GeForce GTX680, are used. All the three computers run a 64-bit version of Windows 7 with the CUDA driver 3.0 as the compiler. Details of the GPU cards are summarized in Table 1.

Table 2: Temperatures used in boundary conditions.

Temperature (°C)	Air	Water	Ground
Summer	28.2	19.5	32.4
Winter	−1.6	2.8	10.3

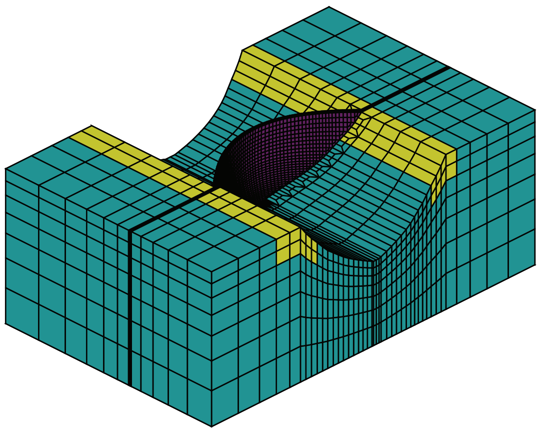
4.2. Test Model. The model of a dam is tested by the following steps.

(i) First, the gravity field is simulated to test the accuracy of the GPU procedure, to get the runtime distribution of different GPU kernels, and to obtain the concerning speedups.

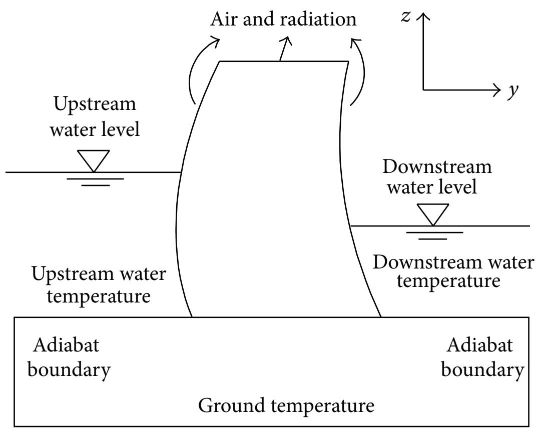
(ii) Then, the thermal field is calculated with a distribution of thermal stress.

(iii) Finally, the fracture field is determined by the superposition of the original gravity and the newly-induced thermal stress.

The model of the dam is presented in Figure 8(a). The boundary conditions for gravity simulation are displacement constraints. The front and back faces ($x^{+/-}$ directions) and the left and right faces ($y^{-/+}$ directions) are all constrained with zero displacements in normal directions. The bottom face (z^{-} direction) is constrained with zero displacements in all directions. The thermal boundary conditions are illustrated in Figure 8(b), and the temperatures used in boundary conditions are presented in Table 2.

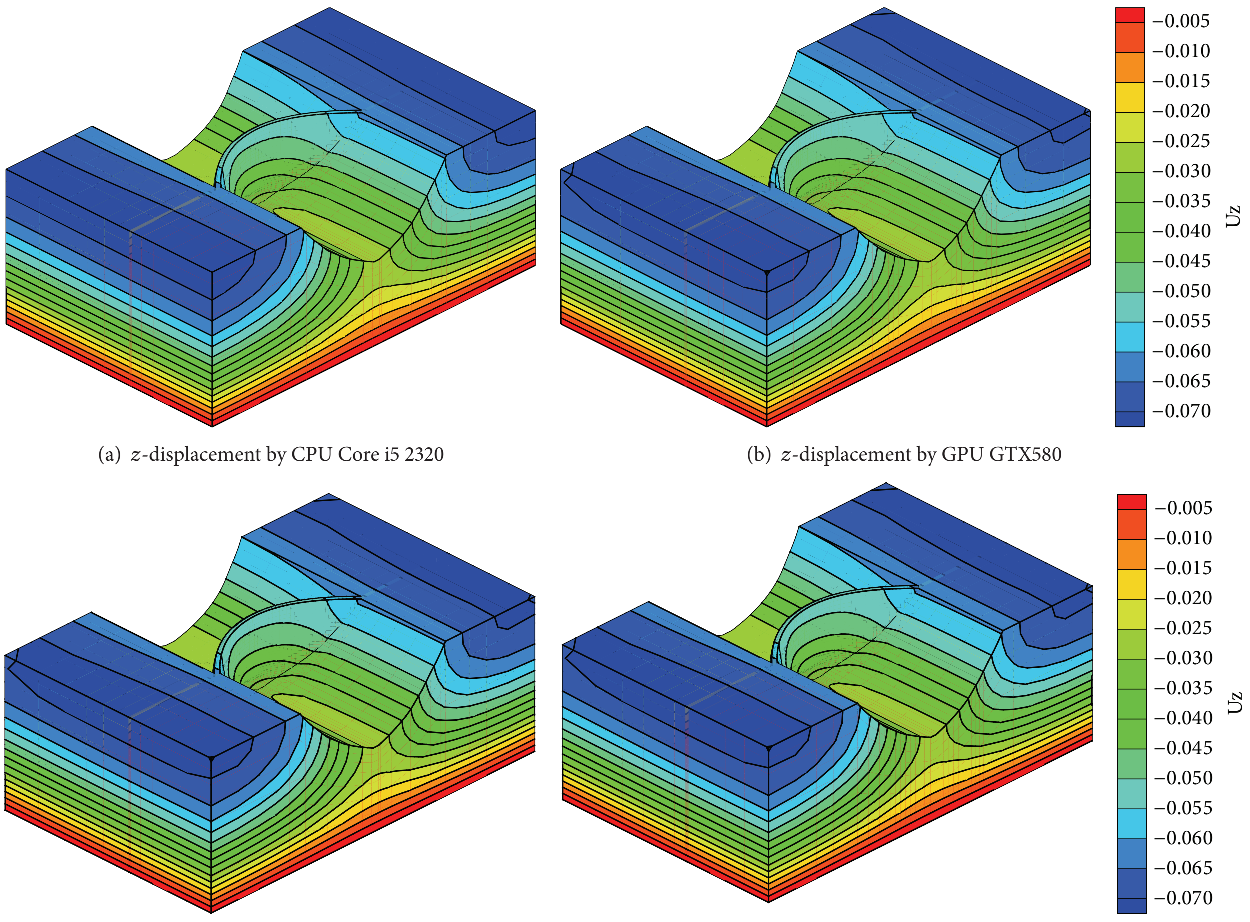


(a) Isotropic view (11166 hexahedron elements)



(b) The thermal boundary conditions

FIGURE 8: The test model of a dam.



(a) z-displacement by CPU Core i5 2320

(b) z-displacement by GPU GTX580

(c) z-displacement by GPU GTX670

(d) z-displacement by GPU GTX680

FIGURE 9: Results from different test platforms.

4.3. GPU Analysis

4.3.1. Accuracy. The results from the CPU and the GPU procedures are not identical though they look the same to each other (see Figure 9). A comparison between the results is illustrated in Figure 10. This comparison shows that the results from different GPUs are the same, but they have a little difference compared with the CPU result. Further, the errors between different procedures are shown in Table 3. The average error is about 2%.

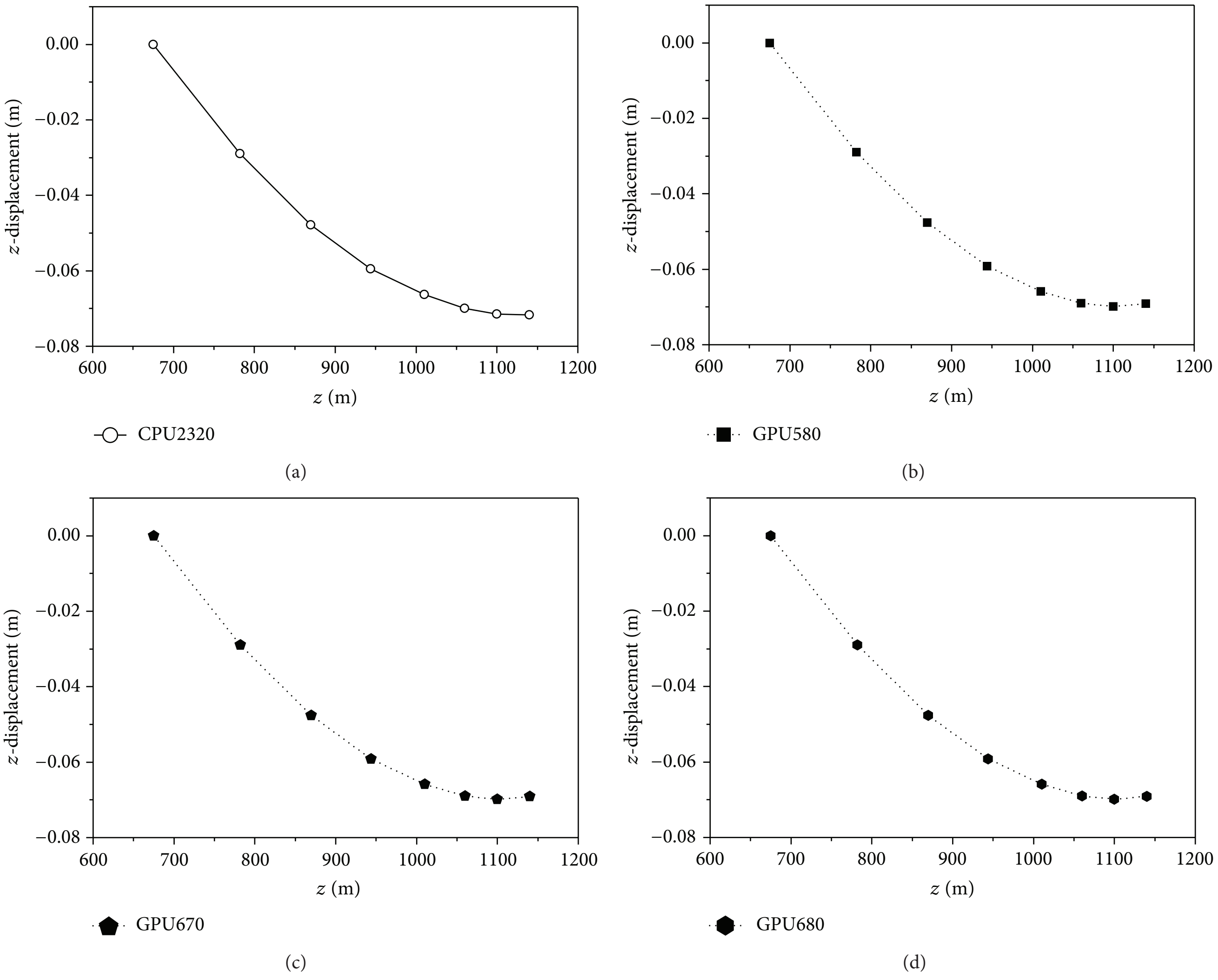
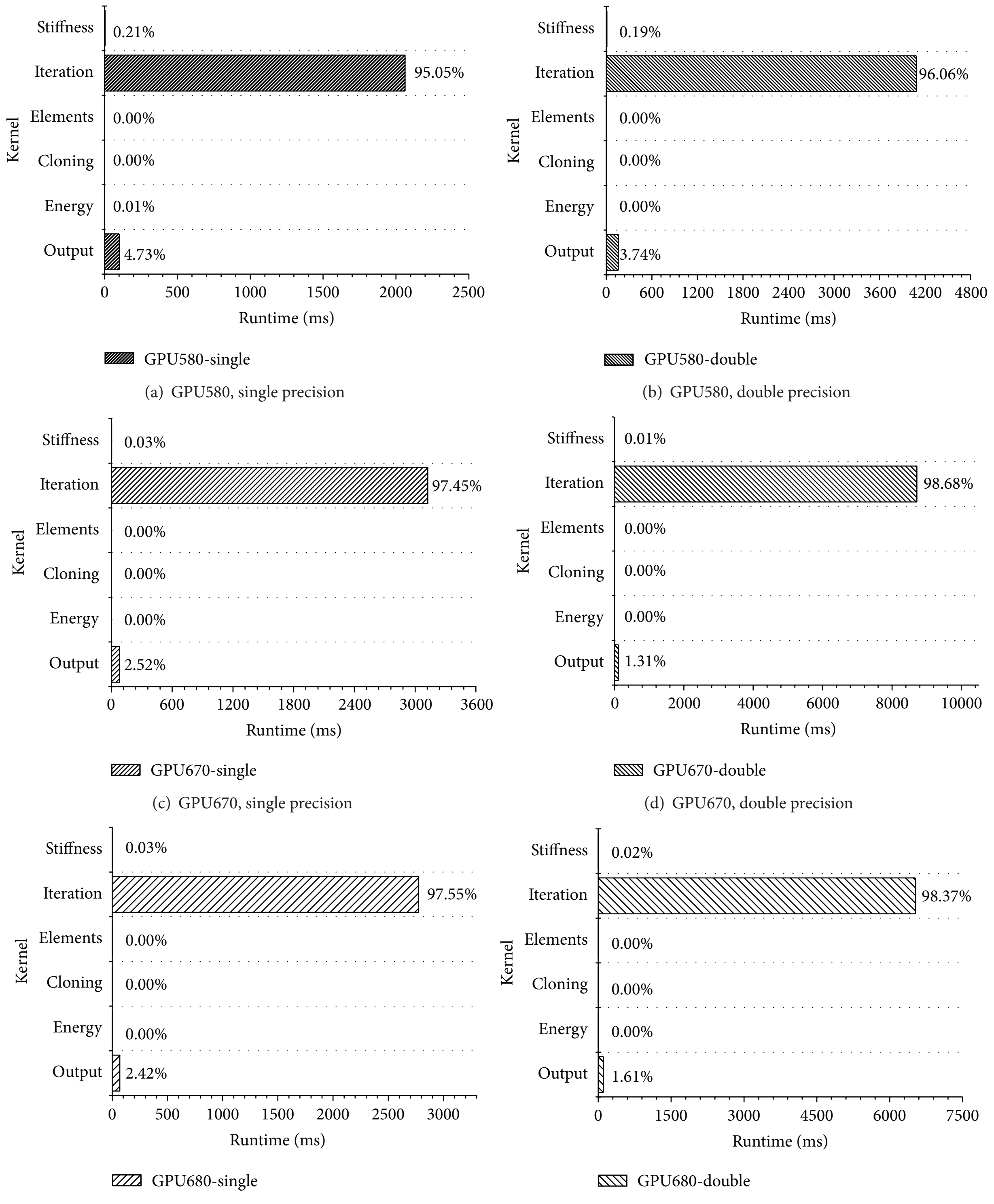


Figure 10: z-displacements at $(x = -578, y = -300)$.

The reason why errors happen is that the parallel algorithm has a different accumulation order from the serial one. The accumulation occurs in the nodal force redistribution function. The CPU procedure accumulates the nodal forces one by one in the order of the elements, while the GPU version has a different order, which depends on the generation of nodal force group. Besides, the Arithmetic Logical Units (ALUs) on CPU and GPU have different relative error bounds.

4.3.2. Runtime. Figure 11 illustrates the runtimes of different GPU kernels. It shows that the iteration kernel of dynamic relaxation takes possession of over 95% of the runtime, while other kernels take possession of that less than 5%. The iteration kernel determines the overall runtime, which means that the iteration speedup can represent the overall speedup to some extent. The second time-consuming is the data output kernel. An average of about 3% of runtime is consumed in data output. When big problems are calculated, this part of runtime can be large. This encourages us to reduce data transfer between *host* and *device*. Other kernels consume little. Their total runtime is less than 1%, which can be neglected in this study.

From another point of view, what makes the runtimes different on different platforms is concerned. Take the runtime of stiffness matrix generation as an example. The runtimes on different platforms are shown in Table 4. On GTX680, the runtime is the least, only 0.828582 ms, nearly the same as the GTX570. The GTX580 is the most time-consuming GPU. The reason why this happens can only be the difference of the numbers of CUDA cores. This indicates that more CUDA cores will reduce the runtime of stiffness matrix generation. However, the number of cores is not the only decisive factor. The processor clocks determine the runtimes of dynamic relaxation iterations, which can be learned from Figure 12. The higher the clock rate is, the less the runtime of iteration is consumed. Figure 13 indicates that newer GPU models are designed for less runtime of output. All the figures presented show that double precision calculations cost more runtimes than single precision ones.



(a) GPU580, single precision

(b) GPU580, double precision

(c) GPU670, single precision

(d) GPU670, double precision

(e) GPU580, single precision

(f) GPU580, double precision

Figure 11: Runtime comparison among different GPU platforms on different computation precisions. “Stiffness” means generating stiffness matrices; “Iteration” is short for dynamic relaxation iteration; “Elements” represents elements and nodal calculation; “Cloning” denotes nodal force redistribution; “Energy” stands for energy calculation; “Output” refers to data output.

Table 3: Errors of z-displacements at $(x = -578,\ y = -300)$.

Node number	Coordinate z (m)	GPU results (m)	CPU results (m)	Errors (%)
1	$1.13500E+03$	$-6.9058E-02$	$-7.1662E-02$	3.63
2	$1.09949E+03$	$-6.9820E-02$	$-7.1442E-02$	2.27
3	$1.05684E+03$	$-6.9004E-02$	$-6.9937E-02$	1.33
4	$9.44115E+02$	$-5.9221E-02$	$-5.9511E-02$	4.87
5	$8.70247E+02$	$-4.7656E-02$	$-4.7809E-02$	0.32
6	$7.81536E+02$	$-2.8922E-02$	$-2.8984E-02$	0.21
7	$0.00000E+00$	$0.0000E+00$	$0.0000E+00$	0.00

Table 4: Runtimes of stiffness matrix generation on different GPUs.

Runtime (ms)	GeForce GTX580	GeForce GTX670	GeForce GTX680
Single precision	4.55253	0.879342	0.828582
Double precision	8.19164	0.901528	0.912056

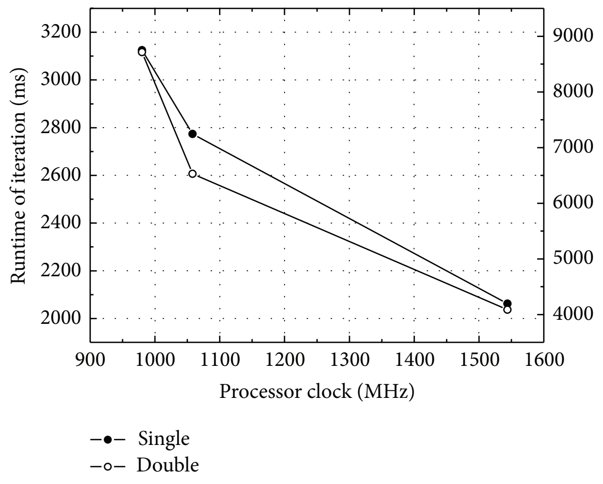


Figure 12: Runtime of iteration versus processor clock. Runtime of iteration decreases with processor clock.

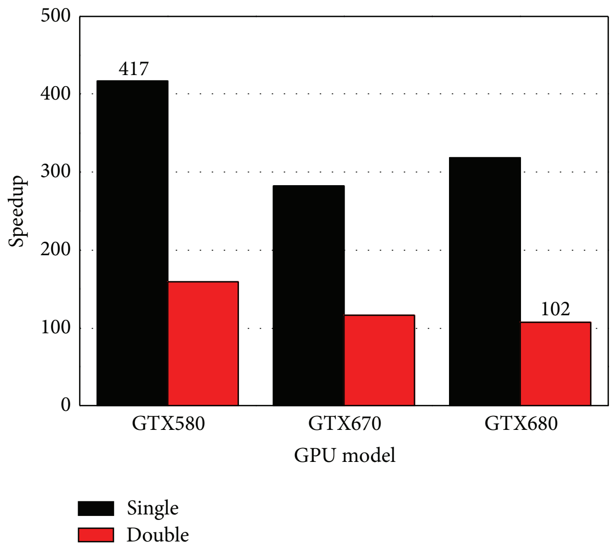


Figure 14: Speedup versus GPU model.

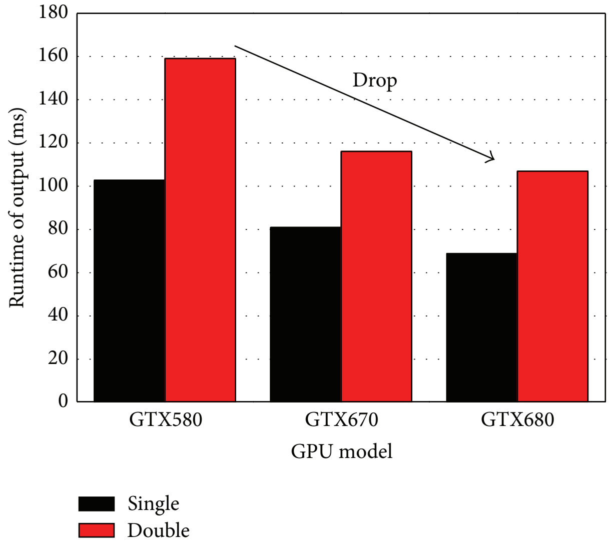


Figure 13: Runtime of output versus GPU model. Newer GPU model consumes less runtime of output.

The runtime analysis helps us to choose a proper GPU model for simulations in both scientific research and engineering applications.

4.3.3. Speedup. In parallel computing, speedup refers to how much a parallel algorithm is faster than a corresponding serial algorithm [30]. Speedup S is defined by the following formula:

$$S = \frac{T_s}{T_p}, \tag{18}$$

where T_s is the runtime of the serial algorithm; T_p is the runtime of the parallel algorithm.

In the tests, we obtain different speedups, which are presented in Figure 14, on different GPU computers using different precisions. The figure shows that speedups range from about 100 to 400. The maximum speedup reaches

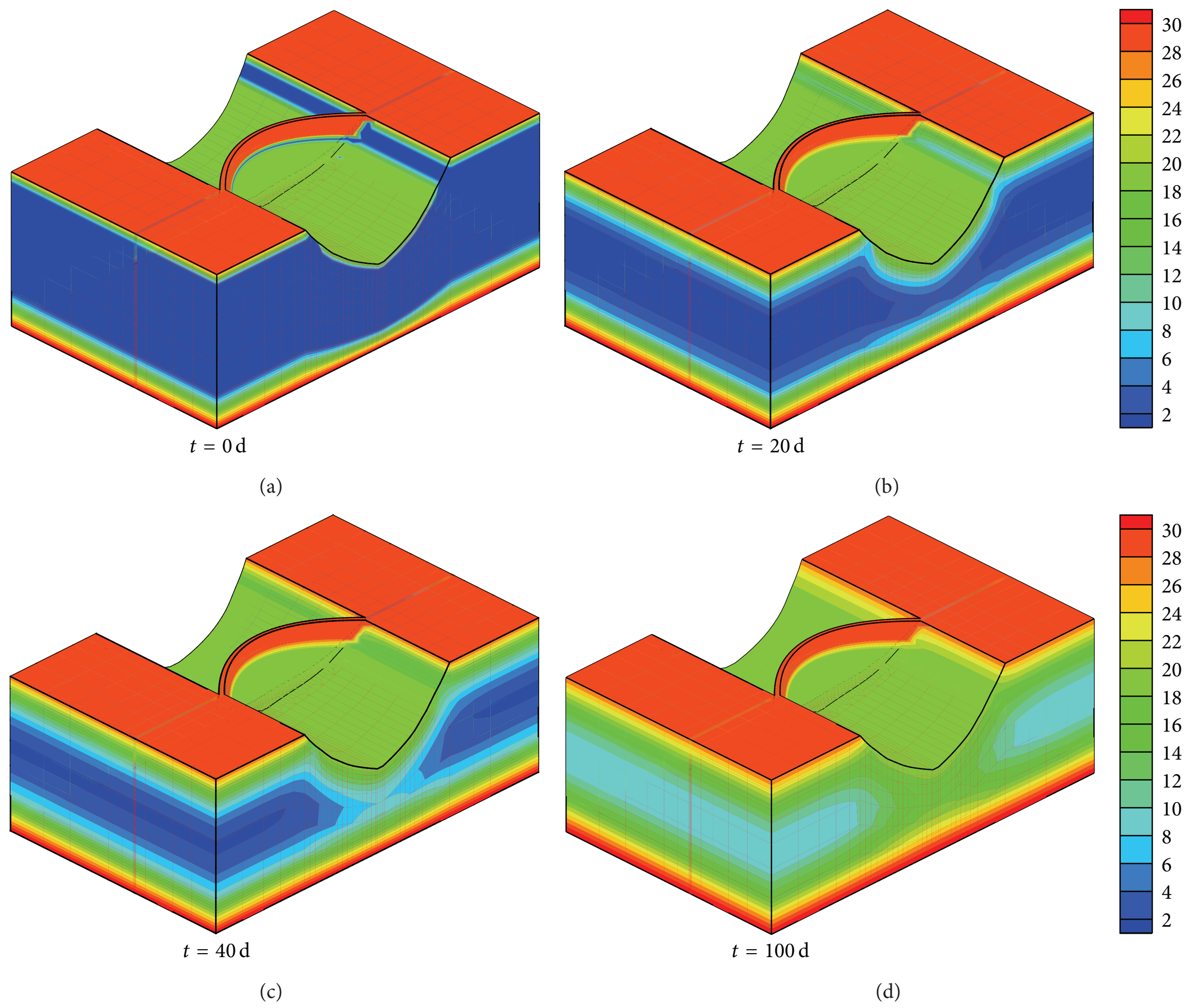


Figure 15: Summer temperature simulation to obtain the steady thermal field. The last picture ($t = 100$ d) shows the steady state of thermal field in summer.

dramaticly 417, while the minimum reaches 102. Following the optimization strategies presented in Section 3.2, we have gained dramatic performance improvements from GPU parallelization. The figure also shows that double precision calculations consume more runtimes than single precision ones. The GTX580 is the fastest GPU among the three GPUs we present.

4.4. Other Analysis. Temperatures in summer and in winter are both simulated, as shown in Figures 15 and 16. We obtain the steady thermal fields in summer (see Figure 15) and in winter (see Figure 16), respectively. The thermal differences between two thermal fields will produce thermal stresses, which may cause cracking in concrete. The crack caused by thermal stresses is a major research topic in future work.

5. Conclusions

In this study, the CDEM is successfully accelerated by using GPUs. In CDEM, global stiffness matrix is not assembled, and an element-by-element strategy is employed to solve the equilibrium equation. This, as well as the performance optimization, enables us to implement an efficient GPU parallel procedure. Detailed tests on accuracy, runtime, and speedup are performed on different GPUs. The results show that dramatic performance improvements are gained from GPU parallelization. A maximum speedup of 417 has been achieved. The GPU parallelization of CDEM makes it more powerful in multifield analysis of complex structures.

Conflict of Interests

The authors do not have any conflict of interests with the content of the paper.

Acknowledgments

The authors would like to gratefully acknowledge the financial support of the National Basic Research Program of China (973 Programs: Grant no. 2010CB731500, Grant no.

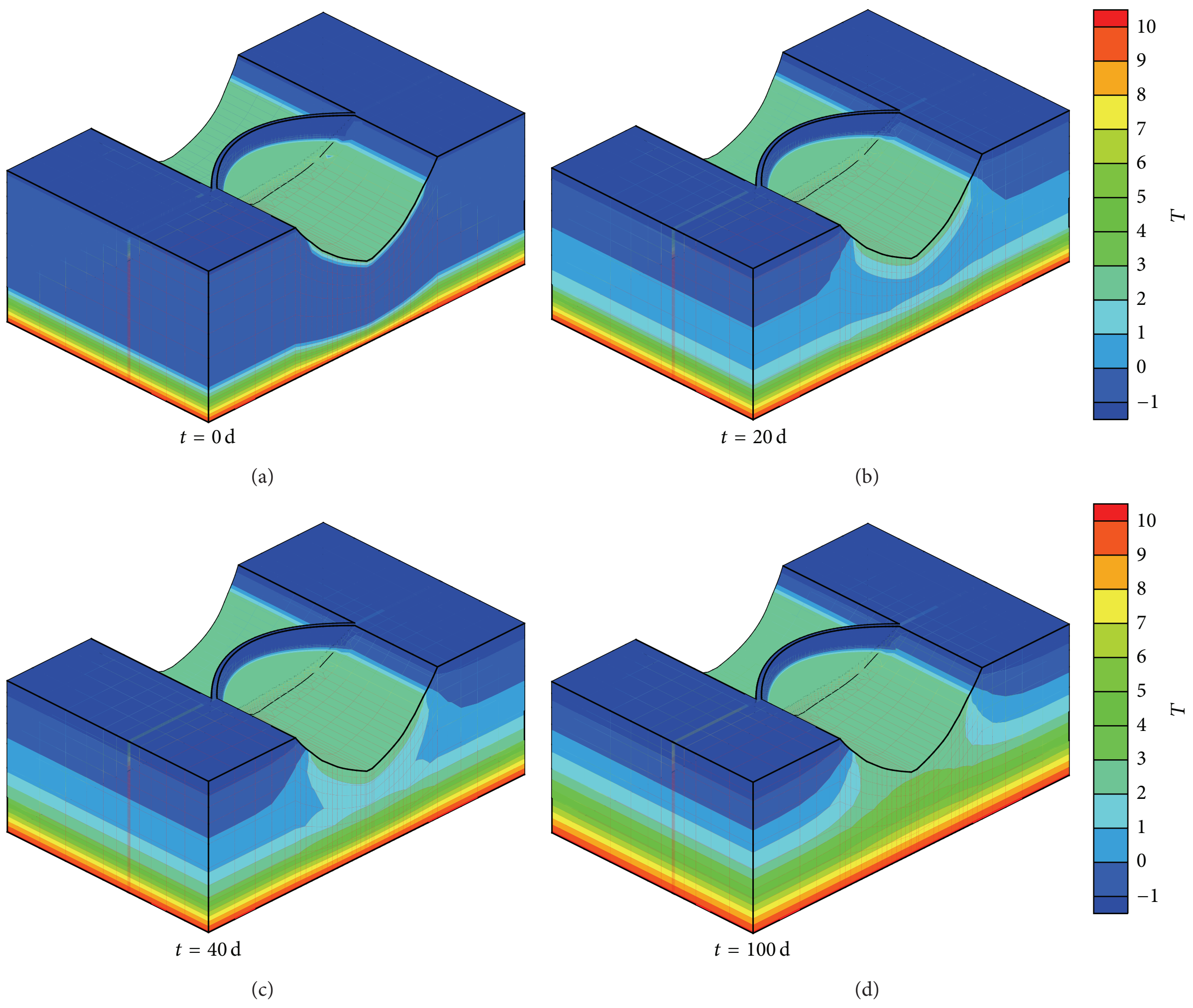


FIGURE 16: Winter temperature simulation to obtain the steady thermal field. The last picture ($t = 100$ d) shows the steady state of thermal field in winter.

2013CB036406, and Grant no. 2013CB035904) and National Natural Science Foundation of China (50909105). Acknowledgments also go to the colleagues/students Chun Feng, Dong Zhou, Qingbo Zhang, and so forth, for their kind help.

References

[1] P. Roca and A. R. Marí, "Nonlinear geometric and material analysis of prestressed concrete general shell structures," *Computers and Structures*, vol. 46, no. 5, pp. 917–929, 1993.

[2] E. Spacone, F. C. Filippou, and F. F. Taucer, "Fibre beam-column model for non-linear analysis of R/C frames: part I. Formulation," *Earthquake Engineering and Structural Dynamics*, vol. 25, no. 7, pp. 711–725, 1996.

[3] J. Rousseau, E. Frangin, P. Marin, and L. Daudeville, "Damage prediction in the vicinity of an impact on a concrete structure: a combined FEM/DEM approach," *Computers and Concrete*, vol. 5, no. 4, pp. 343–358, 2008.

[4] P. Villard, B. Chevalier, B. Le Hello, and G. Combe, "Coupling between finite and discrete element methods for the modelling of earth structures reinforced by geosynthetic," *Computers and Geotechnics*, vol. 36, no. 5, pp. 709–717, 2009.

[5] J. Sziveri, B. H. V. Topping, and P. Iványi, "Parallel transient dynamic non-linear analysis of reinforced concrete plates," *Advances in Engineering Software*, vol. 30, no. 9–11, pp. 867–882, 1999.

[6] M. L. Romero, P. F. Miguel, and J. J. Cano, "A parallel procedure for nonlinear analysis of reinforced concrete three-dimensional frames," *Computers and Structures*, vol. 80, no. 16-17, pp. 1337–1350, 2002.

[7] http://en.wikipedia.org/wiki/Graphics_processing_unit.

[8] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[9] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[10] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-accelerated molecular modeling coming of age," *Journal of*

Molecular Graphics and Modelling, vol. 29, no. 2, pp. 116–125, 2010.

[11] S. D. C. Walsh, M. O. Saar, P. Bailey, and D. J. Lilja, "Accelerating geoscience and engineering system simulations on graphics hardware," *Computers and Geosciences*, vol. 35, no. 12, pp. 2353–2364, 2009.

[12] J. Sainio, "CUDAEASY—a GPU accelerated cosmological lattice program," *Computer Physics Communications*, vol. 181, no. 5, pp. 906–912, 2010.

[13] T. Takahashi and T. Hamada, "GPU-accelerated boundary element method for Helmholtz equation in three dimensions," *International Journal for Numerical Methods in Engineering*, vol. 80, no. 10, pp. 1295–1321, 2009.

[14] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.

[15] G. R. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations on GPU—application to neurosurgical simulation," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 49–52, pp. 3305–3314, 2010.

[16] C. Dick, J. Georgii, and R. Westermann, "A real-time multigrid finite hexahedra method for elasticity simulation using CUDA," *Simulation Modelling Practice and Theory*, vol. 19, no. 2, pp. 801–816, 2011.

[17] J. Xu, H. Qi, X. Fang et al., "Quasi-real-time simulation of rotating drum using discrete element method with parallel GPU computing," *Particuology*, vol. 9, no. 4, pp. 446–450, 2011.

[18] Z. Ma, C. Feng, T. Liu, and S. Li, "A GPU accelerated continuous-based discrete element method for elastodynamics analysis," *Advanced Materials Research*, vol. 320, pp. 329–334, 2011.

[19] J. Liu, Z. Ma, S. Li, and Y. Zhao, "A GPU accelerated Red-Black SOR algorithm for computational fluid dynamics problems," *Advanced Materials Research*, vol. 320, pp. 335–340, 2011.

[20] C. Hori, H. Gotoh, H. Ikari, and A. Khayyer, "GPU-acceleration for moving particle semi-implicit method," *Computers and Fluids*, vol. 51, no. 1, pp. 174–183, 2011.

[21] G. Zhao and N. Khalili, "Graphics processing unit based parallelization of the distinct lattice spring model," *Computers and Geotechnics*, vol. 42, pp. 109–117, 2012.

[22] http://www.nvidia.com/object/cuda_home_new.html.

[23] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, 2009.

[24] S. Li, M. Zhao, Y. Wang, and Y. Rao, "A new numerical method for DEM-block and particle model," *International Journal of Rock Mechanics and Mining Sciences*, vol. 41, supplement 1, pp. 436–440, 2004.

[25] S. Li, X. Liu, T. Liu, and C. Feng, "Continuum-based discrete element method and its applications," in *Proceedings of the UK-China Summer School/International Symposium on DEM*, pp. 147–170, Beijing, China, 2008.

[26] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications.

[27] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-670/specifications.

[28] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications.

[29] NVIDIA, CUDA C Programming Guide, 2012.

[30] http://en.wikipedia.org/wiki/Speedup.