GPU-accelerated regular integration and singular integration in boundary face method

Yuan Li, Jianming Zhang*, Liexiang Yu, Chenjun Lu, Guangyao Li

State Key Laboratory of Advanced Design and Manufacturing for Vehicle Body, College of Mechanical and Vehicle Engineering, Hunan University, Changsha 410082, China

***Corresponding Author: Jianming Zhang**

College of Mechanical and Vehicle Engineering, Hunan University, Changsha, Hunan, 410082, P.R. China

Tel: 86 731 88823061

Email: zhangjianm@gmail.com

Abstract: Graphics processing unit (GPU) is a low-cost, low-power (watts per flop) and very high performance alternative to conventional microprocessors. In addition to their applications in graphics processing, researches are more and more attracted by their potential application in numerical computing due to their powerful ability of parallel computing and rapidly improved programmability. This paper makes a first try of applying GPU to accelerate computation for boundary face method (BFM). The element integration possessing high level of parallelism is a computationally intensive part of the BFM. As a primary step, we have implemented the parallelization of the regular integration and singular integration in CUDA (Compute Unified Device Architecture) programming environment. Comparative computations are made on both NVIDIA GTX 680 GPU and Intel(R) Core(TM) i7-3700K CPU. Results show that, at the same level of accuracy, the speedup of regular integration and singular integration is up to 18.2 and 34.4 respectively.

Keywords: parallel computing, BFM, CUDA, regular integration, singular integration

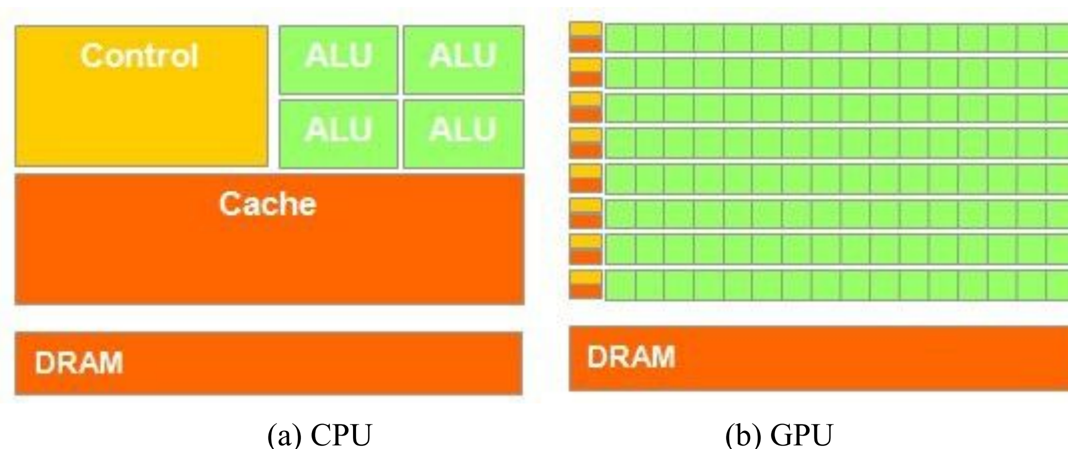
1. INTRODUCTION

In the traditional boundary element method (BEM) (Brebbia et al., 1978; Chen et al., 2001; Wang et al., 2011), boundary elements are employed not only to perform boundary integration and physical variable approximation, but also to approximate the corresponding geometric model. Coarse mesh will cause large geometric errors, and lead to poor computation accuracy. The boundary face method (BFM), which is first proposed by Zhang (Zhang et al., 2008, Zhang, 2012), is a generalization of the conventional BEM and boundary node method (Zhang et al., 2001). In BFM, both variable interpolation and boundary integration are performed in the parametric spaces of the body surfaces. As to the boundary integration, the geometric data of integration point such

as physical coordinates, Jacobians, normal vectors are obtained from the surfaces directly. Therefore, geometric errors are avoided (Qin et al., 2010).

Several works have been published to improve or extend the applicability of the BFM (Wang et al., 2013; Zhou et al., 2013). Due to the increasing scale of problems, a trade-off between the accuracy and efficiency becomes the bottleneck for the application of BFM, and the time for element integration occupies a relatively large proportion to the total analysis time in BFM. To improve the efficiency of the BFM, a parallel integration scheme which is based on GPU is applied in this paper to accelerate the element integration in the BFM. At the primary stage, we have implemented the parallelization of regular integration and singular integration in element integration.

In recent years, GPU comes to be used for general computation rather than only for graphic processing (Nawata et al., 2011). The number of floating-point operations per second and memory bandwidth of GPU are much larger than that of CPU in the same period. Moreover, the price of GPU is relatively lower. Because the GPU is designed for compute-intensive, highly parallel computation, more transistors are devoted to data processing rather than data caching and flow control compared to CPU. As illustrated in figure1, the Arithmetic Logical Units (ALUs) in GPU, which are used for data processing, are more than that in CPU while the storage space of Dynamic Random Access Memory (DRAM) in CPU and GPU are usually same.



(a) CPU (b) GPU

Figure1: The comparison between GPU and CPU hardware structures.

More specifically, the GPUs are especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations (Nvidia Corporation, 2011). In November 2006, NVIDIA introduced CUDA (Nvidia Corporation, 2011; Nvidia Corporation, 2011), a general purpose parallel computing architecture, it allows programmers to develop high performance GPU computing program much more conveniently than before by using the C programming language. Nowadays, CUDA has been widely used in oil exploration (Aksnes et al., 2009), astronomy (Harris et al., 2008), elastodynamic simulation (Wei et al., 2012), computational fluid dynamics (Kakuda et al., 2012; Corrigan et al., 2011; Crane et al.,

2007), biology (Okitsu et al., 2010) and other areas (Mesquita et al., 2009). 10x speedup, even 100x speedup have been achieved in many applications. Cai (Cai et al., 2012) developed parallel explicit finite element sheet forming simulation system based on GPU architecture obtained 27x speedup. Januszewski (Januszewski et al., 2010) accelerated numerical solution of stochastic differential equations with CUDA achieved 675x speedup. Pichel (Pichel et al., 2012) optimized the sparse matrix-vector multiplication, and made the speedup up to 2.6x with the help of CUDA.

In the implementation procedure of BFM, we find that the proportion of time used for integrating the surface cells can be more than 90%. For example, in the first numerical example in section 5, the entire calculation process takes 20.297s when we discretize the model with 15804 nodes, while the time used for integrations is 19.016s. Therefore, in this paper, CUDA is used to accelerate the regular integration and singular integration in BFM. We find that both of them in BFM are very well suited to acceleration on modern GPU. The acceleration and highly accurate achieved on the GPU through two numerical examples.

The remainder of this paper is organized as follows: Section 2 overviews the CUDA. In section 3, we present the boundary integral equations, discretizations, regular integration and singular integration. Section 4 describes the schemes to accelerate regular integration and singular integration in detail. Two numerical examples for evaluating the performance of the developed parallel codes are presented in section 5. The paper ends with conclusions and future work in Section 6.

2. INTRODUCTION OF CUDA

CUDA is NVIDIA's general purpose computation on graphics processing units (GPGPU) model based on C programming language, which can be mastered directly by most people who learned C. The codes written by CUDA are performed in graphics chip without learning specific instructions of the graphics chip or a special structure. GPU consists of several stream multiprocessors (SM). And eight stream processors (SP) are built in a SM. Stream processors are not programmed directly; rather, one writes a CUDA kernel for the GPU. Each kernel consists of a collection of threads arranged into blocks and grids (Aksnes et al., 2009). CUDA provides a large-scale multi-thread architecture making GPU as the coordination processor of CPU. Because GPUs architecture provides hundreds of cores compared with only 1-4 cores in CPUs architecture, thus it is better suited for Single Instruction Multiple Data (SIMD) computations (Liu et al., 2006). The general parallel computing process of CUDA consists of four steps as shown in figure 2: (1) Initialize data on host, (2) Copy data from host to device, (3) Parallel computing, (4) Copy data back to host from device.

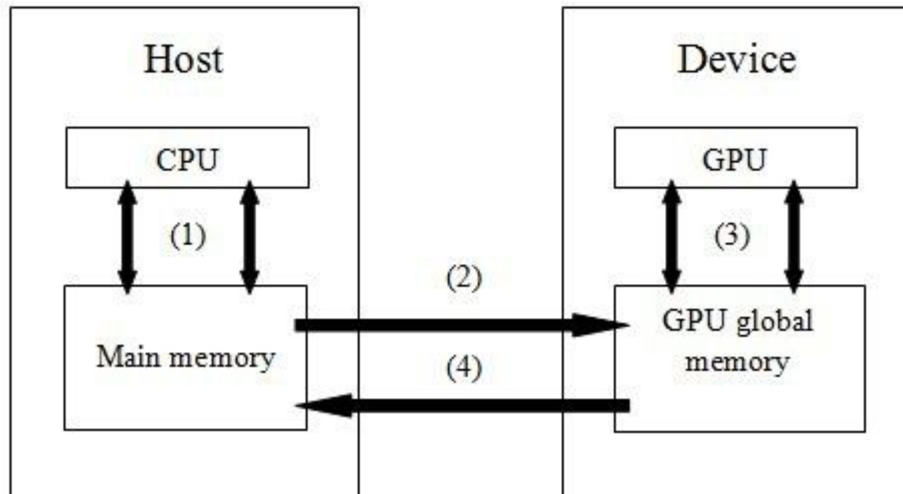


Figure 2: General parallel computing process of CUDA.

GPU threads as illustrated by figure 3 are organized in a grid, and each grid consists of a number of blocks. In G80/GT200 series GPU, a block supports 512/1024 threads. In Fermi architecture (Nvidia Corporation, 2009), the number is up to 1536. Threads in the same block not only execute in parallel, but also communicate with each other through shared memory and barrier. The shared memory and barrier provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism (Nvidia Corporation, 2011). In the actual operation, a block will be divided into smaller units something we called warp (Nvidia Corporation, 2011; Nvidia Corporation, 2011), which is a finer level in the thread's hierarchy. 32 threads form a warp, then the warps are processed by stream processors one by another. The guide of CUDA recommends that the number of threads per block should be chosen as a multiple of the warp size, or better, a multiple of 64 from the viewpoint of performance (Takahashi et al., 2009).

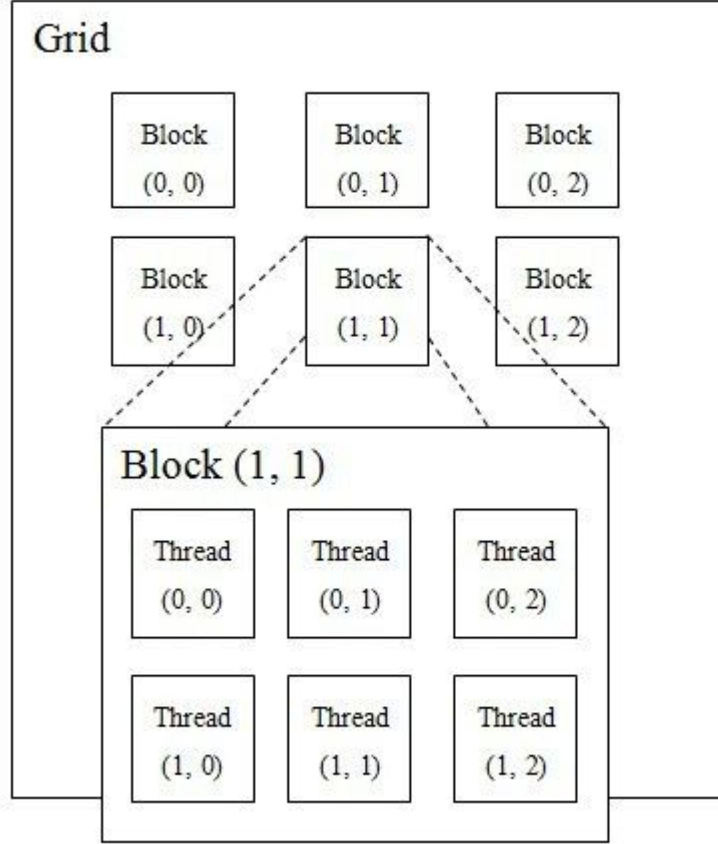


Figure 3: Grid of thread blocks.

3. BOUNDARY INTEGRAL EQUATIONS, DISCRETIZATIONS, REGULAR INTEGRATION AND SINGULAR INTEGRATION

3.1 Boundary integral equations

Firstly the boundary integral equations is given

$$0=\int_{\Gamma}(u(\mathbf{s})-u(\mathbf{y}))q^{s}(\mathbf{s},\mathbf{y})d\Gamma-\int_{\Gamma}q(\mathbf{s})u^{s}(\mathbf{s},\mathbf{y})d\Gamma \quad , \tag{1}$$

where Γ represents the boundary of domain, u and q are prescribed as potential and the normal flux respectively, and $q=\partial u/\partial n$; y and s stand for the source point and the field point on the boundary respectively. $q^{s}(\mathbf{s},\mathbf{y})$ and $u^{s}(\mathbf{s},\mathbf{y})$ are fundamental solutions. For 3-D potential problems,

$$u^{s}(\mathbf{s},\mathbf{y})=\frac{1}{4\pi}\frac{1}{r(\mathbf{s},\mathbf{y})}, \tag{2}$$

$$q^{s}(\mathbf{s},\mathbf{y})=\frac{\partial u^{s}(\mathbf{s},\mathbf{y})}{\partial n}=\frac{1}{4\pi r^{2}(\mathbf{s},\mathbf{y})}\frac{\partial r(\mathbf{s},\mathbf{y})}{\partial n(\mathbf{s})}. \tag{3}$$

3.2 Discretizations

At first the boundary of domain should be discretized into M boundary cells and N corresponding interpolation nodes. As to any node, either u or q is known. u and q on the boundary can be approximated by the following formulas:

$$u(s) = u(u,v) = \sum_{k=1}^{N} N_k u_k \\ q(s) = u(u,v) = \sum_{k=1}^{N} N_k q_k \quad , \tag{4}$$

where, u and v are 2-D parameter coordinates of boundary parameter surface, 3-D physical space coordinates (x,y,z) can be represented by polynomials: $x = x(u,v)$, $y = y(u,v)$, $z = z(u,v)$, N_k is the shape function. Substituting Eqn. (4) into Eqn. (1), we have

$$0 = -\sum_{j=1}^{M} \int_{\Gamma_j} q^s(\mathbf{s},\mathbf{y}) \sum_{k=1}^{N} (N_k(\mathbf{s}) - N_k(\mathbf{y})) u_k d\Gamma + \sum_{j=1}^{M} \int_{\Gamma_j} u^s(\mathbf{s},\mathbf{y}) \sum_{k=1}^{N} N_k(\mathbf{s}) q_k d\Gamma . \tag{5}$$

Eqn. (5) can be formed in a matrix form as

$$\mathbf{H}u = \mathbf{G}q , \tag{6}$$

where

$$H_{ik} = \sum_{j=1}^{M} \int_{\Gamma_j} q^s(s,y_i)(N_k(s) - N_k(y_i)) d\Gamma , \tag{7}$$

$$G_{ik} = \sum_{j=1}^{M} \int_{\Gamma_j} u^s(s,y_i) N_k(s) d\Gamma . \tag{8}$$

3.3 Regular integration and singular integration

The first term on the right hand side of Eqn. (5) is regular in any case. Therefore, the regular Gaussian integration scheme can be used to evaluate it over each cell. As to the second term on the right hand side of Eqn. (5), if the distance between s and y is far enough, we consider that is regular. Take a quadrilateral cell shown in figure 4 as an example. L is the longest line in the quadrilateral cell, d is the distance between point y and s. If $d > Lk$ (k is a scale factor, its value is 4.0 in this paper), the cell can be treated as regular cell.

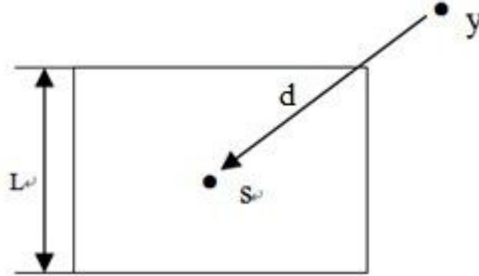
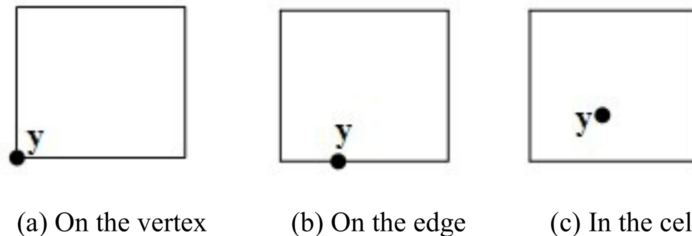


Figure 4: Quadrilateral cell.

Considering the fundamental solutions of integral equation in potential problems, when y (source point) and s (field point) belong to the same cell, as shown in figure 5, the cell is treated as singular cell, and $r(\mathbf{s},\mathbf{y}) \to 0$, $u^s(\mathbf{s},\mathbf{y}) \to \infty$, so the second term on the right hand side of Eqn. (5) will become weakly singular.



(a) On the vertex (b) On the edge (c) In the cell

Figure 5: The position of source point in quadrilateral cell.

4. ACCELERATION METHOD WITH CUDA

4.1 Regular integration

We use the CUDA to compute matrix **H**, **G** and the geometric data at Gaussian integration points such as Jacobians, normals, shape functions for regular integration. In CPU-based code, **H** and **G** are computed serially. Figure 6 shows the algorithm: in the first loop, there are N cells, each cell contains a number of field points(three field points in liner triangle cell). Geometric data are computed in this loop. In the second loop, **H** and **G** are computed. One item of **H** and **G** can be obtained in one cycle. When the second loop terminates, then, repeat the same procedure for other cells in the first loop.

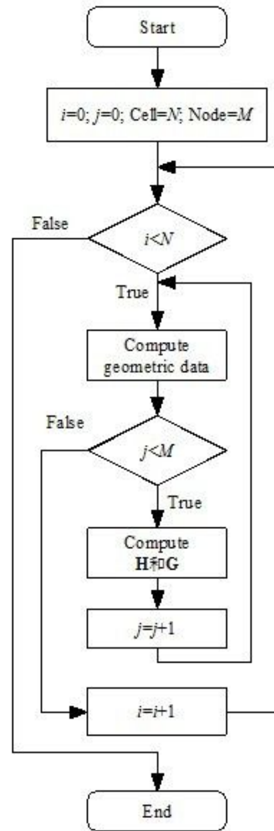


Figure 6: Flowchart of CPU computing.

We develop GPU-based code with CUDA as figure 7 shown. Initially we transfer data of integration patches and cell data such as 3D physical coordinate and 2D parameter coordinate of field points to GPU memory. N stands for the cell number. Then we apply the tid_in_grid to locate the thread position in the whole grid. Each thread executing the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable (Nawata et al., 2011). The blockIdx.x and threadIdx.x come from CUDA runtime, blocksize is defined as the length of block in x-direction. In this paper, we chose one-dimensional grid and one-dimensional block. $Q1$ stands for the cell number. The first kernel is assigned to compute the geometric data

at Gaussian integration points. N threads execute in parallel in the first kernel, one thread computes the geometric data for one cell. Then we allocate the G_Host and H_Host in main memory, their dimensions are $N \times N$. $Q2$ stands for the number of regular integration points for each cell. G_Device and H_Device which located in GPU memory are used for storing the results of the second kernel which computes **G** and **H**. The second kernel is called by CPU for N times. One row of items in **G** and **H** can be obtained in one second kernel call by using the geometric data computed by the first kernel. When a kernel computing procedure is finished, the results stored in G_Device and H_Device are transferred back to CPU.

```
01: Transfer data of patchs and cell to GPU memory
02: Q1, N←Cell number
    /// kernel computing geometric data
03: tid_in_grid ← blockIdx.x * blocksize + threadIdx.x
04: if tid_in_grid < Q1 then
05:     compute geometric data.
06: Allocate G_Host [N*N], H_Host [N*N] in main memory
07: Allocate G_Device [N], H_Device [N] in GPU memory
    ///kernel computing G and H.
08: for i=0 to N do
09:     tid_in_grid ← blockIdx.x * blocksize + threadIdx.x
10:     Q2←number of regular integration points
11:     if tid_in_grid < Q2 then
12:         compute G_ Device and H_ Device
13:     end if
14:     Transfer G_Device and H_Device from GPU memory to CPU memory
15: end for
```

Figure 7: Algorithm of regular integration with GPU-based code.

4.2 Singular integration

For singular integration, the singular cell would be subdivided into M ($M > 1$) cells to increase the gauss integration points and insure the computing accuracy. Figure 8 shows the subdivision of a triangular cell, and the cell is subdivided into five cells. The subdivision of a singular cell is dependent on the location of the source point. Detailed rules of the subdivision can refer to paper of Qin et al. (2010).

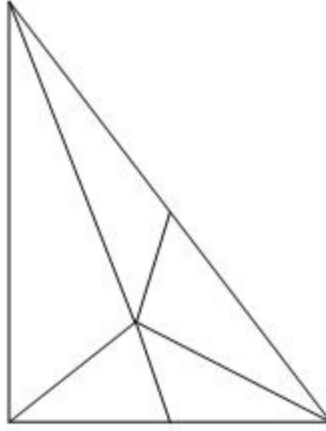


Figure 8: The triangular cell subdivision.

Figure 9 shows the algorithm of singular integration in GPU. As the same to regular integration, we also transfer data of patches and cell data to GPU memory in the first step. Q and N stand for the cell number. Cell subdivision is executed parallel in GPU. We assign Q threads in the grid, and one cell is subdivided into M cells in one thread. The loop in line 09~12 is used to compute the geometric data at Gaussian integration points. Results obtained in line 11 will be added to G_Device and H_Device in each step of loop. Finally G_Device and H_Device are transferred to G_Host and H_Host which have been allocated in main memory.

```
01: Transfer data of patchs and cell to GPU memory
02: Q, N←Cell number
03: P←number of field point in cell
04: Allocate G_Device [P*P], H_Device [P*P] in GPU memory
05: tid_in_grid←blockIdx.x * blocksize + threadIdx.x
06: if tid_in_grid < Q then
07:     cell subdivision
08:     M←the number of patch subdivided
09:     for i=0 to M do
10:         compute Jacobians etc.
11:         compute G_Device and H_Device
12:     end for
13: end if
14: Transfer G_Device and H_Device from GPU memory to CPU memory
```

Figure 9: Algorithm of singular integration in GPU.

5. NUMERICAL RESULTS

The GPU-based code of BFM developed by CUDA has been tested for two types of 3-D geometrical objects: a cuboid and a foundation bed. The first object is used to compare the GPU-based code with the CPU-based code on accuracy and efficiency performance. We adopt the

foundation bed as an example to demonstrate that our code can be used for objects with different shapes. In this paper, field variables are approximated by linear triangular elements. In order to assess the accuracy of the method, we have used the following cubic analytical field:

$$u = x^3 + y^3 + z^3 - 3yx^2 - 3xz^2 - 3zy^2 . \quad (9)$$

The GPU-based code is tested on the Computing platform listed in table 1.

Table 1: Computing platform.

OS	Windows 7 64bit with 16GB main memory
CPU	Intel(R) Core(TM) i7-3700K CPU 3.5GHZ
GPU	NVIDIA GTX 680 GPU with NVIDIA driver version 301.32, it consists of 192 streaming multiprocessors (SM), 2GB global memory. The compute capability is 3.0
Complier	Visual studio 2008, CUDA toolkit 4.2.9 64bit

The performance of GPU implementation is compared against the serial-oriented code of the CPU implementation. Speedup is obtained by dividing the time of CPU-based codes computing by the time of GPU-based codes computing.

5.1 Dirichlet problem on a cuboid

The size of cuboid we used is $4 \times 10 \times 30$ mm, meshed by liner triangular elements. Figure 10 shows the cuboid discretization for the BFM analysis with 476 elements and 1428 nodes.

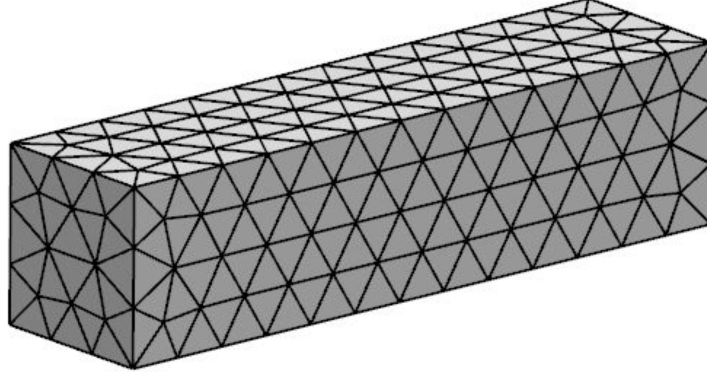


Figure 10: The BFM mesh for cuboid.

5.1.1Regular integration

Table 2 shows the computing time and errors of results of both CPU-based code and GPU-based code, and the last column lists the speedup of regular integration. CPU_Time and GPU_Time represent the computing time cost of the CPU-based code and the GPU-based code, respectively. Compute_GPU reprsents the time of CUDA kernel computing, Translate_GPU stands for the time for copying data between the main memory and the GPU global memory, Compute_GPU plus Translate_GPU equals GPU_Time. It is obvious that Translate_GPU occupies a large proportion of GPU_Time, this is because the number of double data which is transferred to GPU shown in figure 7 is up to $2 \times N \times N$. Therefore, the bandwidth limitations of GPU we used slow the growth of speedup when the node number is large enough.

Table 2: Result of regular integration for cuboid.

Node	*CPU_ Time (s)*	*Compute_ GPU (s)*	*Translate_ GPU (s)*	*GPU_ Time(s)*	*CPU_ Errors (%)*	*GPU_ Errors (%)*	*Speedup*
1428	0.145	0.000	0.049	0.049	0.09357	0.09357	2.960
2412	0.356	0.000	0.0505	0.0505	0.0776	0.0776	7.049
3384	0.713	0.000	0.0575	0.0575	0.06322	0.06322	12.400
6636	3.121	0.078	0.1175	0.1955	0.04639	0.04639	15.964
9012	5.878	0.185	0.1635	0.3485	0.04086	0.04086	16.867
13152	14.663	0.250	0.5715	0.8215	0.03691	0.03691	17.849
15804	20.297	0.310	0.805	1.115	0.03571	0.03571	18.201

Figure 11 exhibits the speedup of GPU-based code. As figure 11 shown, at first the speedup grows rapidly with the node number increases, and then it grows slowly when the node number is large enough. When the node number reaches 15804, the speedup can be up to 18.2. It is easy to conclude that GPU-based code behaves much better than CPU-based code considering the computing efficiency. Figure 12 compares errors of the results of GPU-based code with that of CPU-based code. It is shown that the accuracy of the GPU computation is good enough for regular integration in BFM.

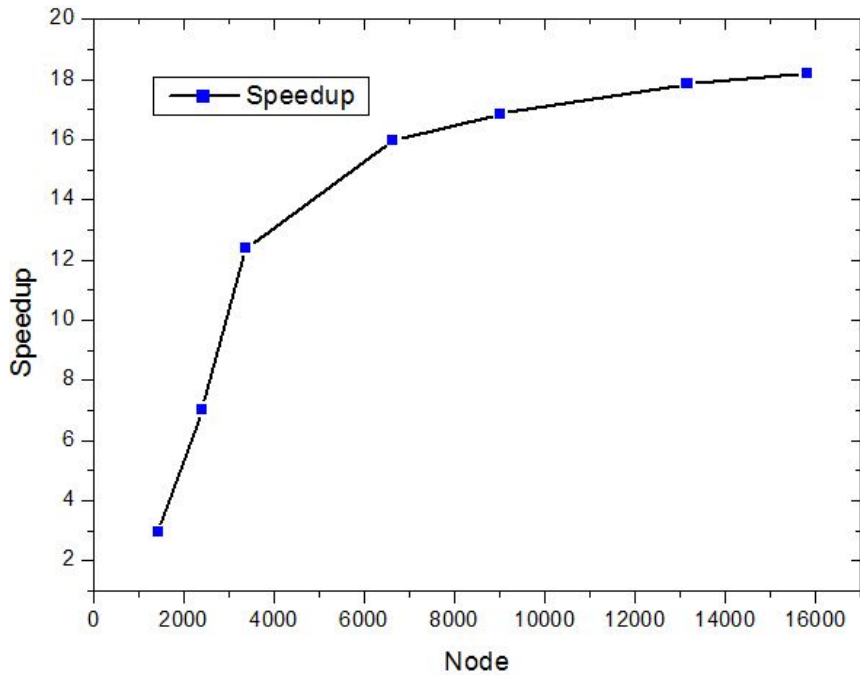


Figure 11: Speedup of regular integration for cuboid.

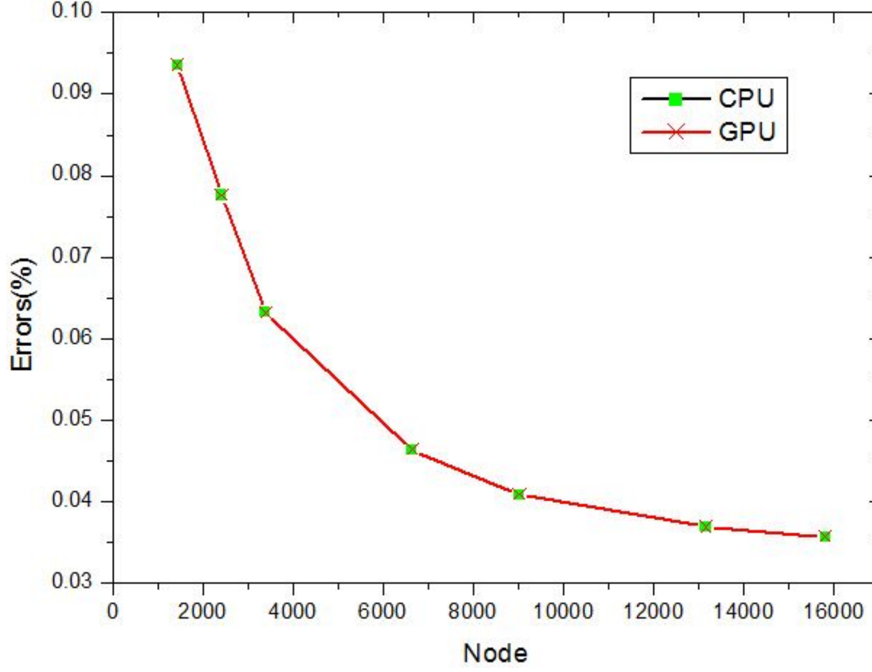


Figure 12: Errors of regular integration for cuboid.

5.1.2 Singular integration

Table 3 shows the computing time and errors of results of both CPU-based code and GPU-based code, and the last column lists the speedup of singular integration. From the table bellow, we can see that the Compute_GPU always stays at 0s even when the node number reaches 15804, and the GPU_Time is fully occupied by Translate_GPU. The reason is that the timing precision in C++ is millisecond, and the computational complexity in GPU is too low, the time of Compute_GPU lower than 0.001s will be set to 0.000s.

Table 3: Result of singular integration for cuboid.

Node	*CPU_ Time(s)*	*Compute_ GPU(s)*	*Translate_ GPU(s)*	*GPU_ Time(s)*	*CPU_ Errors(%)*	*GPU_ Errors(%)*	*Speedup*
1428	0.053	0.000	0.008	0.008	0.09357	0.09357	6.625
2412	0.110	0.000	0.010	0.010	0.0776	0.0776	11.000
3384	0.183	0.000	0.013	0.013	0.06322	0.06322	14.077
6636	0.321	0.000	0.016	0.016	0.04639	0.04639	20.061
9012	0.425	0.000	0.017	0.017	0.04086	0.04086	25.000
13152	0.621	0.000	0.021	0.021	0.03691	0.03691	29.571
15804	0.861	0.000	0.025	0.025	0.03571	0.03571	34.440

Figure 13 exhibits the speedup of singular integration for cuboid. According to the picture, we can see that the speedup increases almost linearly with the growth of the node number. The number of double data which is transferred to GPU shown in figure 7 is $2\times N\times P\times P$ (P stands for number of field points in one cell, and N stands for cell number), it is far smaller than the number in regular integration. Therefore, the speedup of singular integration is much higher than

that of regular integration without the bandwidth limitations. When the node number reaches 15804, the speedup can be up to 34.4. The accuracy of GPU computing of the singular integration is equal to that of the regular integration as shown in figure 12.

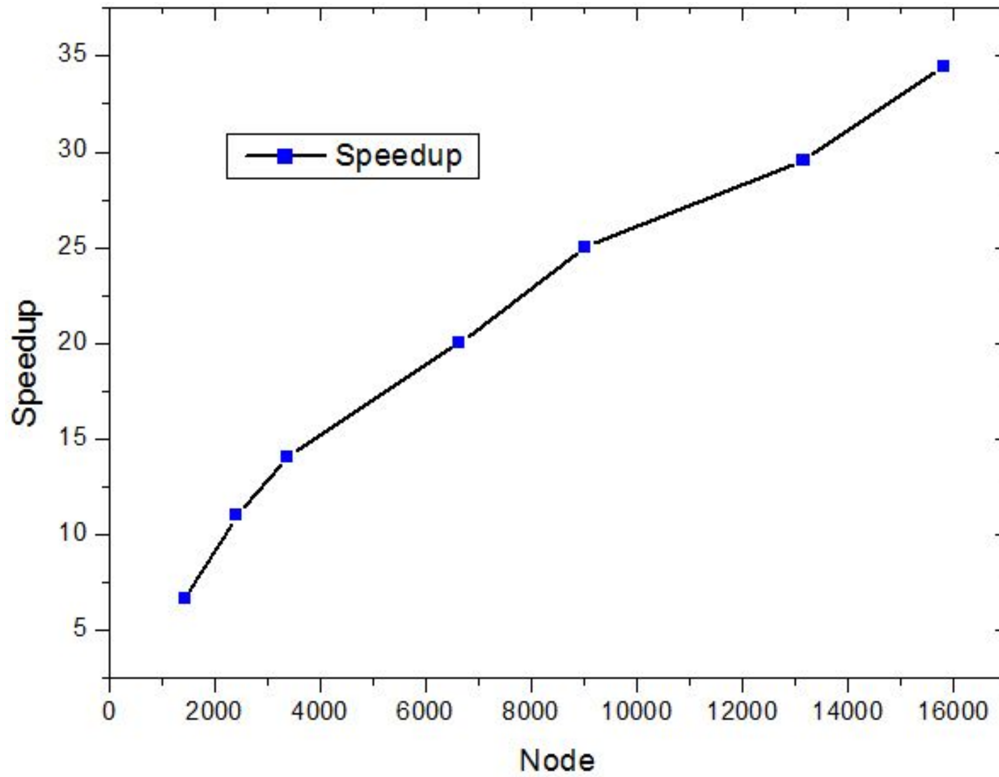


Figure 13: Speedup of singular integration for cuboid.

5.2 Dirichlet problem on a foundation bed

Computation on a foundation bed is presented as the second example. In figure 14, the body is meshed with 846 linear triangular elements and 2538 nodes. In the following part, we will remesh the model in order to observe the relationship between speedup and the number of nodes.

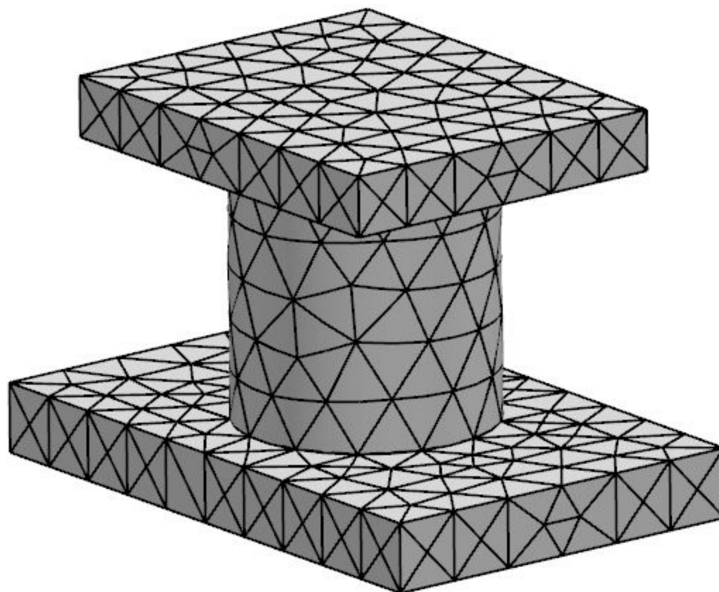


Figure 14: The BFM mesh for foundation bed.

Because the time cost of GPU computing of foundation bed is same to that of cuboid, the table like table 2 is not given again. As figure 15 and figure 16 shown, when the node number reaches 15882, the speedup of regular integration and singular integration is up to 17.9 and 34.1 respectively. Figure 17 shows the errors of results for the foundation bed. It is seen that the numerical results obtained by both codes are in good agreement with the analytical solution.

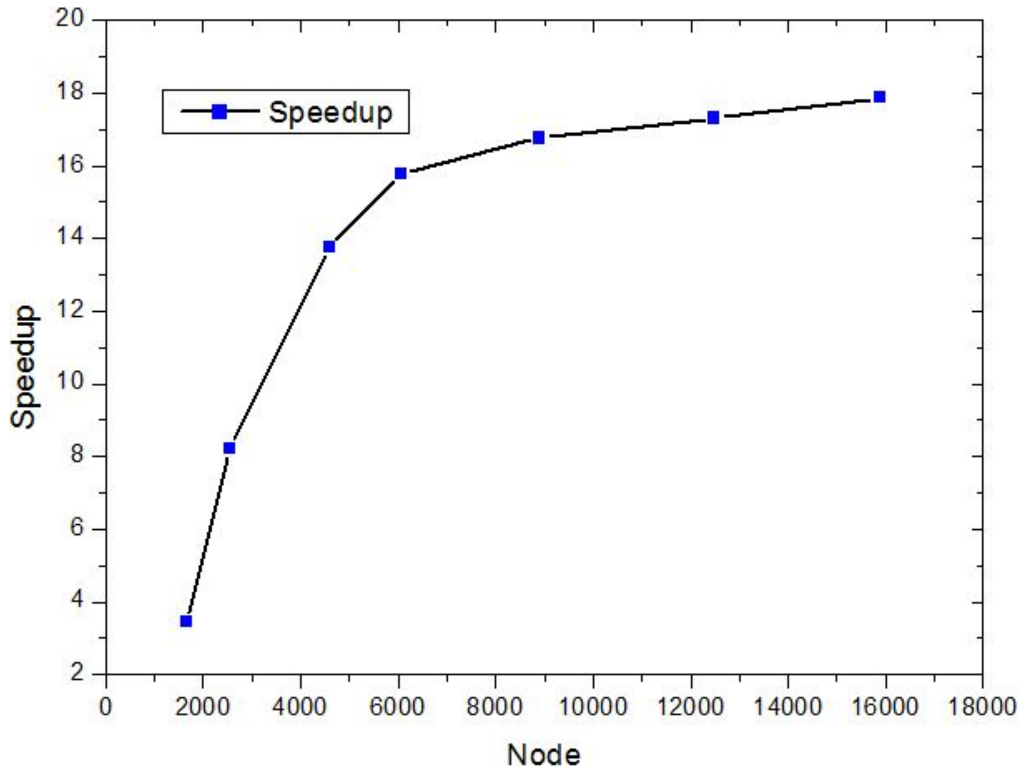


Figure 15: Speedup of regular integration for foundation bed.

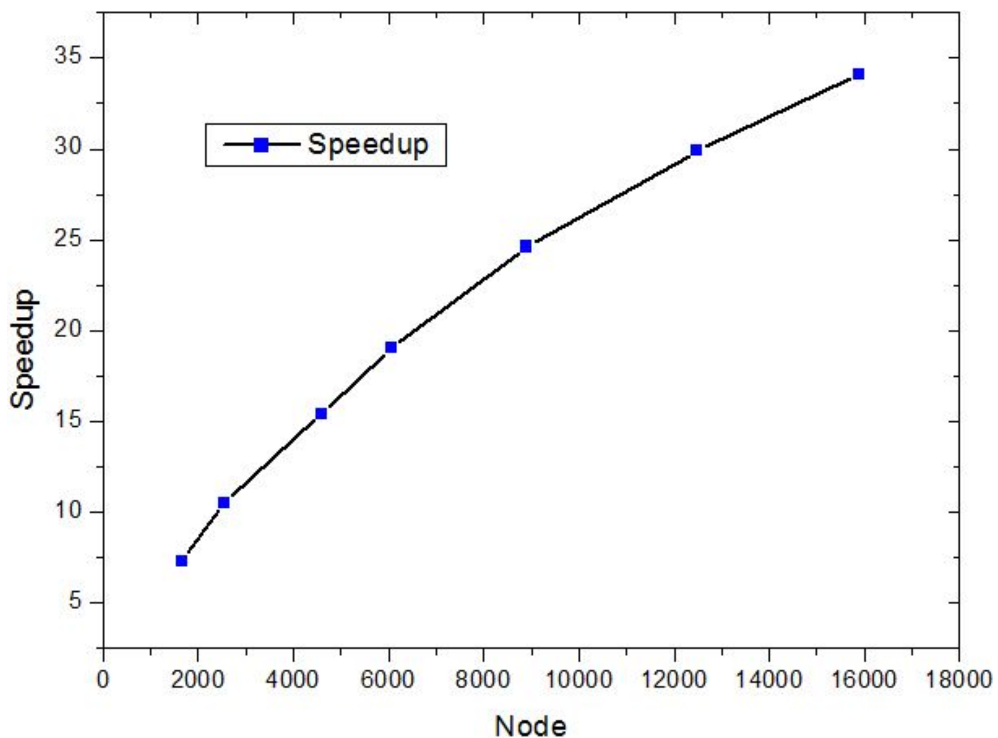


Figure 16: Speedup of singular integration for foundation bed.

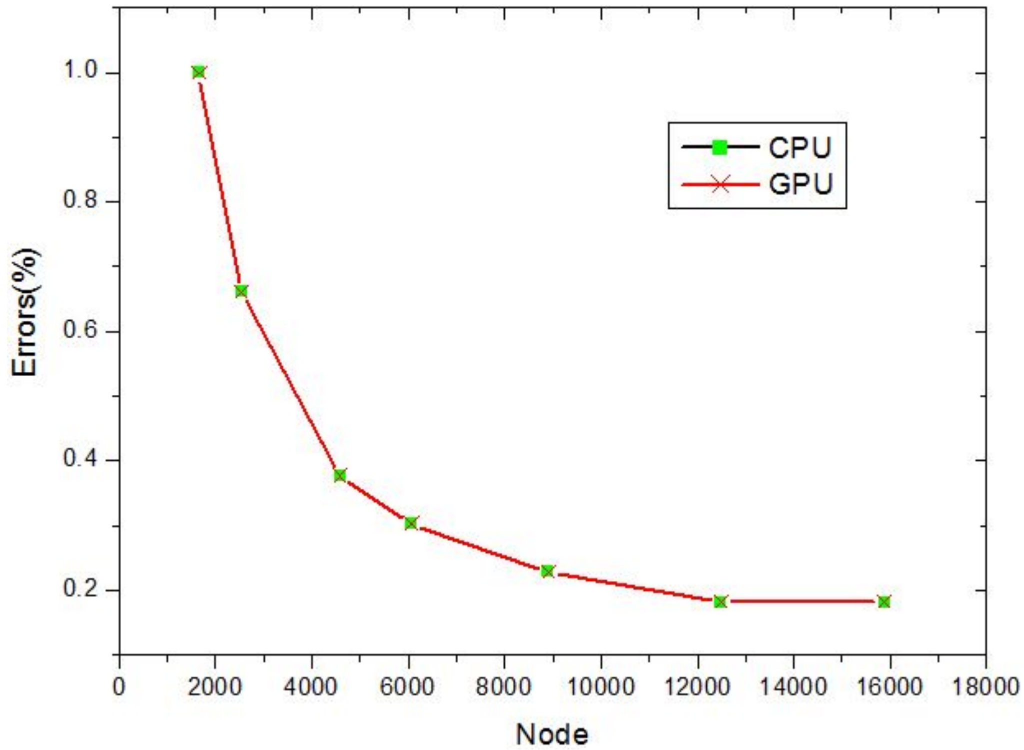


Figure 17: Errors of numerical results for foundation bed.

6. CONCLUSIONS

In this paper, we have developed the GPU-based code for regular integration and singular integration in boundary face method (BFM), and demonstrated the suitability of a parallel CUDA-based hardware platform for BFM. The speedup of regular integration and singular integration is up to 18.2 and 34.4 respectively, when the node number reaches 15882. As to regular integration, because of the large scale of input data, the time cost of data transmission between host and device will affect the efficiency of the code seriously. Therefore the bottleneck of improving the efficiency of the program is bandwidth, which is related to the hardware performance of GPU. The computational accuracy of the GPU-based code is comparable with that of the CPU-based code for double-precision floating-point computation. As future work we plan to use CUDA to accelerate the near singular integration and LU-decomposition of large-scale linear equations in BFM. With the lower cost of high performance GPU, there is no doubt that the development of stream processing technology for general-purpose computing has just started and its potential is surely not yet fully revealed.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation of China under grant numbers 11172098.

REFERENCES

Aksnes, E.O., Hesland, H. & Elster, A.C. 2009, GPU techniques for porous rock visualization. technical report.

Brebbia, C.A. 1978, *The Boundary Element Method for Engineers*, London, Pentech Press.

Cai, Y., Li, G.Y., Wang, H., Zheng, G. & Lin, S. 2012, "Development of parallel explicit finite element sheet forming simulation system based on GPU architecture", *Advances in Engineering Software*. Vol. 45, No. 1, pp. 370–379.

Chen, X.L. & Liu, Y.J. 2001, "Thermal stress analysis of multi-layer thin films and coatings by an advanced boundary element method", *Computer Modeling in Engineering & Sciences.* Vol. 2, No. 3, pp. 337-339.

Corrigan, A., Camelli, F., Löhner, R. & Wallin, J. 2011, "Running unstructured grid-based CFD solvers on modern graphics hardware", *International Journal for Numerical Methods in Fluids*. Vol. 66, No. 2, pp. 221–229.

Crane, K., Llamas, I. & Tariq, S. 2007, "Real-time simulation and rendering of 3d fluids", *GPU Gems*. Vol. 3, pp. 663–675.

Harris, C., Haines, K. & Staveley-Smith, L. 2008, "GPU accelerated radio astronomy signal convolution", *Experimental Astronomy*. Vol. 22, No. 1, pp. 129–142.

Januszewski, M. & Kostur, M. 2010, "Accelerating numerical solution of stochastic differential equations with CUDA", *Computer Physics Communications*. Vol. 181, No. 1, pp. 183–188.

Kakuda, K., Nagashima, T., Hayashi, Y., Obara, S., *et al.* 2012, "Particle-based Fluid Flow Simulations on GPGPU Using CUDA", *Computer Modeling in Engineering & Sciences.* Vol. 88, No. 1, pp. 17-28.

Liu, Y., Huang, W., Johnson, J. & Vaidya, S. 2006, "GPU accelerated smith-waterman", *Computer Science*. Vol. 3994, pp. 188–195.

Mesquita, E., Labaki, J. & Ferreira, L. 2009, "An implementation of the Longman's integration method on graphics hardware", *Computer Modeling in Engineering & Sciences.* Vol. 51, No. 2, pp. 143-167.

Michalakes, J. & Vachharajani, M. 2008, "GPU acceleration of numerical weather prediction", *Parallel and Distributed Processing.* Vol. 18, No. 4, pp. 531–548.

Nawata, T. & Suda, R. 2011, "APTCC: Auto parallelizing translator from C to CUDA", *Procedia Computer Science*. Vol. 4, pp. 352–361.

Nvidia Corporation. http://www.Nvidia.corn/cuda

Nvidia Corporation 2009, NVIDIA's next generation CUDA compute architecture. Fermi.

Nvidia Corporation. 2011, CUDA C programming guide. Version 4.0.

Nvidia Corporation. 2011, CUDA C best practices guide. Version 4.0.

Okitsu, Y., Ino, F. & Hagihara, K. 2010, "High-performance cone beam reconstruction using CUDA compatible GPUs", *Parallel Computing*. Vol. 36, No. 2-3, pp. 129-141.

Pichel, J.C., Rivera, F.F., Fernandez, M. & Rodriguez, A. 2012, "Optimization of sparse

matrix-vector multiplication using reordering techniques on GPUs", *Microprocessors and Microsystems*. Vol. 36, No. 2, pp. 65–77.

Qin, X.Y., Zhang, J.M., Li, G.Y., Sheng, X.M., Song, Q. & Mu, D.H. 2010, "An element implementation of the boundary face method for 3D potential problems", *Engineering Analysis with Boundary Elements*. Vol. 34, pp. 934–943.

Takahashi, T. & Hamada, T. 2009, "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.

Wang, H.T. & Yao, Z.H. 2011, "A fast multipole dual boundary element method for the three-dimensional crack problems", *Computer Modeling in Engineering & Sciences.* Vol. 72, No. 2, pp. 115-147.

Wang, X.H., Zhang, J.M., Zhou, F.L. & Zheng, X.S. 2013, "An adaptive fast multipole boundary face method with higher order elements for acoustic problems in three-dimension", *Engineering Analysis with Boundary Elements.* Vol. 37, pp. 114–152.

Wei, Y.X., Wang, Q.F., Wang, Y.J. & Huang, Y.B. 2012, "Optimizations for elastodynamic simulation analysis with FMM-DRBEM and CUDA", *Computer Modeling in* Engineering & Sciences. Vol. 86, No. 3, pp. 241-273.

Zhang, J.M. 2012, http://www.5aCAE.com

Zhang, J.M., Qin, X.Y., Han, X. & Li, G.Y. 2008, "A boundary face method for potential problems in three dimensions", *International Journal for Numerical Methods in Engineering.* Vol. 80, pp. 320–337.

Zhang, J.M. & Yao, Z.H. 2001, "Meshless Regular Hybrid Boundary Node Method", *Computer Modeling in Engineering & Sciences*. Vol. 1, No. 1, pp. 1-12.

Zhou, F.L., Xie, G.Z., Zhang, J.M. & Zheng, X.S. 2013, "Transient heat conduction analysis of solids with small open-ended tubular cavities by boundary face method", *Engineering Analysis with Boundary Elements.* Vol. 37, pp. 542-550.