Copyright © 2008 Tech Science Press CMES, vol.25, no.2, pp.115-131, 2008

Finite Element Analyses of Dynamic Problems Using Graphics Hardware

Atsuya Oishi[1] and Shinobu Yoshimura[2]

Abstract: This paper describes the finite element analyses of dynamic problems using graphics hardware. The graphics hardware, known as GPU that is an acronym of Graphics Processing Unit, was first developed only for processing 3D computer graphics. However it has obtained both flexible programmability using a high-level shader programming language such as OpenGL Shading Language (GLSL), and has also obtained fast numerical processing ability of over 160 GFLOPS that is much faster than CPU. In this paper, GPU is utilized for the finite element analyses of dynamic problems. Two different computational tasks in the dynamic finite element analyses are implemented to the GPU. One is the construction of element stiffness/mass matrices and the other is the calculation of time integration based on an explicit scheme. Fundamental formulations of the implementations are described in detail, and their basic performance is tested through sample analyses. The results indicate that GPU can perform dynamic finite element analyses faster than CPU.

Keyword: Graphics Hardware, Graphics Processing Unit, Finite Element Method, Dynamic Problem, Explicit Scheme, Element Matrix.

1 Introduction

There has always been strong demand for high level 3D computer graphics in various fields such as games, animation and CAD/CAM [Kawai (2004)], and it has promoted very rapid progress of graphics processing hardware on PCs. Graphics hardware formerly had only hardware-acceleration for several fixed functions. Today's graphics hardware, however, has come to be programmable by users and is called GPU, Graphics Processing Unit, after CPU, Central Processing Unit [Fernando (2004), Pharr (2005)].

As GPUs were specially designed to process characteristic tasks of computer graphics fields where a very large amount of pixels should be concurrently processed by almost the same operations among them, they are suitable for parallel processing of a large amount of data. Latest GPUs are some of the most powerful computational chips that are broadly available, in other words they are "commodities". In addition, the rate of improvement in the computational power of GPUs is much higher than that of CPUs.

The flexible programmability and strong computational power of today's GPUs make it reasonable to use them for general numerical processing of various applications other than graphics. This is called GPGPU, General Purpose GPU [Fernando (2004), Pharr (2005), Owens, Luebke, Govindraju, Harris, Krüger, Lefohn and Purcell (2005), Rumpf and Strzodka (2006)].

Several GPGPU researches have been done in the field of CAE (Computer Aided Engineering) related simulations. Though applications based on FDM (Finite Difference Method) [Harris, Baxter III, Scheuermann and Lastra (2003)], particle based method such as SPH (Smoothed Particle Hydrodynamics) [Müller, Charypar and Gross (2003)] and LBM (Lattice Boltzmann Method) [Li, Wei and Kaufman (2003)], mass - spring method [Georgii and Westermann (2005)] are popular, applications based on the FEM (Finite Element Method) [Rumpf and Strzodka (2001), Wu and Heng (2004)] are very few in the GPGPU field. The FEM for solid or fluid problems does not show essential parallelism in contrast to particle based methods, and also it often tackles problems with complex geometry using unstructured

[1] Univ.Tokushima, Tokushima, JAPAN.
[2] Univ.Tokyo, Tokyo, JAPAN.

meshes, which results in irregular, unstructured sparse matrix that is hard to get high performance in GPU computing. However, the FEM is the most important scheme in the CAE-related fields because of its flexibility in representing both complex geometry of an analysis domain and complex boundary conditions required in those fields. Therefore, from a practical viewpoint, it is worth implementing several procedures of the FEM on GPUs in order to make full use of their computational power.

In this paper, two key processes of the finite element analyses of dynamic problems are implemented on GPUs. One is the construction of element stiffness/mass matrices for almost one million elements, and the other is the calculation of time integration based on an explicit scheme for large scale problems, e.g. approximately one million DOFs (Degrees of Freedom). Comparing the computational nature of each process, these two processes are totally different and needs different computational techniques to be implemented on GPUs. The fundamental formulations of these implementations are described in detail, and their basic performance is tested through sample analyses.

2 Graphics Hardware

2.1 Graphics Processing Unit

In our study, GeForce 7800GTX of Nvidia Corporation is used as the target GPU, whose theoretical peak computation speed is over 160GFLOPS that is more than ten times higher than that of Pentium 4 630 CPU. Both the GPU and its graphics memory are mounted on a graphics board, i.e. an extension board to be installed into the graphics extension bus of PC, such as AGP, PCI-Express. Though high-speed, wide-band graphics memory ranging from 256MB to 512MB is usually available, it is impossible to extend its size in contrast to the expandability of the main memory of CPU up to 4GByte.

As GPU was originally developed only for computer graphics, one must use the so-called graphics API, such as OpenGL [OpenGL ARB (2005)] and DirectX [Jones (2004)], to control the present GPU. Though DirectX is the proprietary API of Microsoft, OpenGL was originally developed by Silicon Graphics and is in the control of the OpenGL ARB (Architecture Review Board), an independent consortium consists of a lot of graphics-related companies. OpenGL is now supported on various systems, such as Xwindow on Linux/BSD and Microsoft Windows.

A program that describes operations to be done by GPU is called a shader. The shader is written in the shader language, such as Cg [Fernando and Kilgard (2003)], HLSL[St-Laurent (2005)] and GLSL (OpenGL Shading Language) [Rost (2006)]. Cg is developed by Nvidia and is used with API, OpenGL or DirectX. GLSL is the standard for OpenGL.

2.2 Numerical Computation on GPU

Textures, data structure similar to two dimensional array for CPU and usually used for storing image to be mapped onto surfaces of objects, are used to store input/output data during execution of numerical computation on GPU. An element of a texture, called a texel, can store up to four 32bit floating point data in the case of the texture of RGBA type, where RGB is the acronym of three kinds of colors, Red, Green and Blue, and A is transparency value, Alpha.

Operations on the data in the input textures are programmed in the shader and executed by rendering the data onto the output texture of the same size as input textures using the shader. Though operations only on a single texel are described in the shader, the operations are executed over all texels in the texture in a SIMD (Single Instruction Stream and Multiple Data Stream) parallel processing manner [Dowd (1993)]. For example, if 512x512 RGBA input texture is rendered onto 512x512 RGBA output texture using the shader that outputs the each input data multiplied by 2, almost one million (512x512x4) multiplications by 2 are to be executed.

Results calculated by a shader are written into the frame buffer as default, which is directly connected to the display and can store only 8bit data per color. In order to write and store 32bit data, the shader output should be written into the specially arranged offscreen buffer. In the OpenGL, FBO extension (GL_EXT_Framebuffer_Object) [Astle (2006), Boreskov (2006)], which is supported in OpenGL 2.0 or later, makes it possible for a shader to output 32bit data into textures. Using textures as offscreen output buffer also makes it easier to reuse the results of one shader as the input for another shader. Basic configuration for numerical computation by a shader on the GPU is illustrated in Fig. 1. In this figure, several texels are colored to make things clear: data in the colored texels in the input texture are processed by the shader and corresponding results are written into the texels of the same color in the output texture.

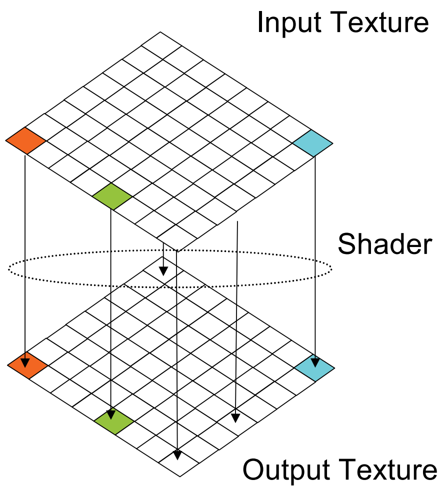


Figure 1: Numerical Computation in GPU

GPUs have very poor I/O capabilities. In standard implementations of numerical computation on GPUs, the input data for GPU computing are prepared by CPU and transferred to the GPU through graphics bus. The output data of GPU are also transferred to CPU memory through the graphics bus to be saved into disks or to be prepared for following computation on the CPU or the GPU. Therefore the overhead due to the data transfer should be taken into account to achieve better performance. The GPU can do one job, while the CPU does another job, in other words the GPU and the CPU can work concurrently. This suggests that we should regard the PC with the GPU embedded as the multiprocessor system consisting of two processors, the CPU and the GPU. So, several techniques to achieve better performance in the parallel processing PC clusters, such as using large computation granularity and making transfer frequency as low as possible, also apply to the CPU-GPU system.

Numerical computation on GPU by shaders has the following limitations:

(1) Precision: Calculation in double precision, is not supported by today's GPU hardware.

(2) Shader Length: There is a limitation on the number of instructions in a shader. There also exist some limitations related to the complexity of shaders and the number of variables available at once.

(3) Number of Input Data: There exists a limitation on the number of textures accessible from a shader at once. This results in setting a practical limit on the number of input data that a shader can access.

(4) Number of Output Data: The number of calculated results written into the output texture by a shader should not be more than four per texel.

(5) Data Transfer Rate: Data transfer rate from the GPU to the CPU through graphics bus is slow. It is slower than the transfer rate in the reverse direction, from CPU to GPU.

3 Calculation of Element Stiffness/Mass Matrices Using GPU

3.1 Element Stiffness/Mass Matrices

Using the finite element method [Bathe (1996)], discretization of the equilibrium equation of a static structural problem with respect to space dimensions results in the following equation,

$$[K]\{u\} = \{f\} \tag{1}$$

where $\{u\}$ is a displacement vector, $[K]$ a global stiffness matrix and $\{f\}$ an external force vector. Discretization of the motion equation of a dynamic structural problem with respect to space dimensions leads to the following equation,

$$[M]\{\ddot{u}\}+[K]\{u\}=\{f\} \tag{2}$$

where $\{\ddot{u}\}$, $\{u\}$ are an acceleration vector and a displacement vector respectively, $[M]$,$[K]$ a global mass matrix and a global stiffness matrix respectively, and $\{f\}$ an external force vector.

The global stiffness/mass matrix is constructed by summing up all the element stiffness/mass matrices as follows:

$$[K]=\sum_{e=1}^{n}[k^e] \tag{3}$$

$$[M]=\sum_{e=1}^{n}[m^e] \tag{4}$$

where n is the number of matrices. The element stiffness/mass matrices are calculated by the following equations respectively,

$$[k^e]=\int_{v^e}[B]^T[D][B]\,dv \tag{5}$$

$$[m^e]=\int_{v^e}[N]^T\rho\,[N]\,dv \tag{6}$$

where $[N]$, $[B]$, $[D]$, ρ, v^e are a matrix of shape functions, a strain-displacement matrix, a stress-strain matrix, a mass density and volume of an element, respectively.

For some 3D elastic solid elements, the number of nodes per element, the number of DOFs per element and the size of the element matrix are specified in Table 1. The linear tetrahedron element and the linear hexahedron element are illustrated in Fig. 2(a) and 2(b) respectively.

Table 1: 3D Solid Elements

Element Type	Nodes	DOFs	Matrix Size
Linear Tetrahedron	4	12	144 (12×12)
Quadratic Tetrahedron	10	30	900 (30×30)
Linear Hexahedron	8	24	576 (24×24)
Quadratic Hexahedron	20	60	3600 (60×60)

There are two kinds of mass matrix: a consistent mass matrix and a lumped mass matrix. The consistent mass matrix is the one calculated by Eq.

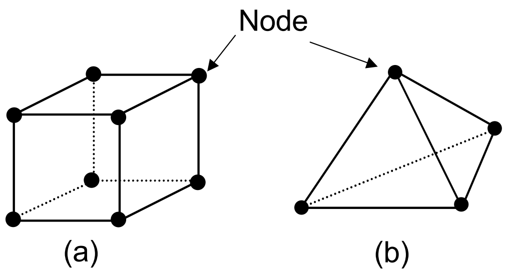


Figure 2: Solid Elements

(6). The lumped mass matrix is made from the consistent mass matrix by setting each diagonal component as the sum of the correspondent row of the consistent mass matrix and non-diagonal elements as 0. The *ij*-th component of the lumped mass matrix m^L_{ij} is calculated from the *ij*-th component of the consistent mass matrix m_{ij} as follows:

$$m^L_{ii}=\sum_j m_{ij},\quad m^L_{ij}=0\quad (i\neq j) \tag{7}$$

The numerical integration in Eqs. (5), (6) is performed using Gauss-Legendre method in which the integrated value is approximated by the sum of the value of the integrand multiplied by the weight factor at several integration points. This means the workload of the numerical integration is proportional to the number of integration points.

3.2 Implementation to GPU

3.2.1 Input Data

Coordinate values of nodes that define element and material properties are required for computing the element stiffness/mass matrices of the corresponding element. These input data should be set in several textures to be accessed from a shader. For example, to compute the element stiffness matrix of the 8-noded linear hexahedral solid element (Fig.2(a)), totally 24 node coordinates (three coordinates per node) and two material properties, i.e. Young's modulus and Poisson's ratio, are required. To compute the element mass matrix, a mass density instead of Young's modulus and Poisson's ratio is required as material property. This results in 26 input data for stiffness matrix and 25 input data for mass matrix. Hereafter, to make description clear and compact, we focus on the case of 8-noded linear hexahedral solid element.

The proposed GPU computing of element matrices reads necessary data from several input textures, and outputs results to the output buffer / texture of the same size as input textures. Both input textures and the output texture have one to one mapping with elements. Every data in a texel in each input texture, which corresponds to an element, is processed by the shader, and its results are written into the texel of the same texture coordinates as the input texel in the output texture. This is illustrated in Fig. 3. For input textures, RGBA-type textures that can store up to four 32bit floating-point data per texel are used. The sizes of input textures are set to make the number of texels in each input texture just more than that of total elements: e.g. the texture size should be more than 1000x1000 for totally one million elements. The number of input textures should be set just more than $\lfloor (number\ of\ input\ data)/4 + 0.9 \rfloor$, where $\lfloor\rfloor$ means the floor function, so that they can store all the input data in the textures.

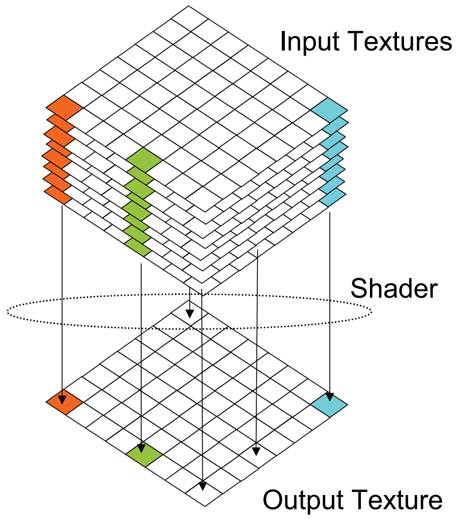


Figure 3: Element Matrix Computation by Shader

3.2.2 Output Data

The proposed GPU computation of an element matrix writes the results, i.e. matrix components, into the texel corresponding to the element in the output texture as shown in Fig. 3. However one texel in the output texture of RGBA-type can store only four floating point data while an element matrix consists of more than a hundred of components. So a lot of shaders are needed to output all the components of one complete element matrix. To output 576 (=24×24) components of the stiffness matrix of an 8-noded linear hexahedral solid element, for example, 144 (=576/4) shaders are needed. Each shader, from the shader001 to the shader144, outputs only four components of the element matrix as follows:
shader001 outputs $k_{1,1}, k_{1,2}, k_{1,3}, k_{1,4}$,
shader002 outputs $k_{1,5}, k_{1,6}, k_{1,7}, k_{1,8}$,
...
shader143 outputs $k_{24,17}, k_{24,18}, k_{24,19}, k_{24,20}$,
shader144 outputs $k_{24,21}, k_{24,22}, k_{24,23}, k_{24,24}$.

A pseudo code for Shader001, for example, is shown as follows:

```
{ //Pseudo Code for Shader001
 Several Texture Arguments
  {
   Calculation of Shape Functions
   Calculation of Derivatives of Shape Functions
   Calculation of Jacobian Matrix
   Calculation of Inverse Jacobian Matrix
   Calculation of [B] matrix
   .............(some calculations).................
   gl_FragColor.r = $k_{11}$//Output $k_{11}$
   gl_FragColor.g = $k_{12}$//Output $k_{12}$
   gl_FragColor.b = $k_{13}$//Output $k_{13}$
   gl_FragColor.a = $k_{14}$//Output $k_{14}$
  }
}
```

Codes for other shaders are almost the same as the code above, except for the output part. Each of the shaders outputs four different components.

While computing element matrices on a CPU needs looping over all elements, it is not necessary to loop over all elements in computing them on a GPU. This is because processing every texels by shaders, where the total number of texels is equivalent to that of total elements, equals to looping over all elements. Looping over shaders that output different components of matrix respectively, however, is mandatory on the present implementation because each shader can output only a small portion of the whole components of element matrix. Loop count of the shader loop depends not on the number of elements but on the number of matrix components, which depends on element type.

The following pseudo code shows the usual implementation of the computation of element matrices on a CPU.

```
for(k: all elements){
  .........................
  for(i,j: all components of a matrix){
    output $k_{ij}$ of the k-th element
  }
}
```

And the following pseudo code shows our implementation of the computation of element matrices on a GPU.

```
for(i,j: all components of a matrix){
  for($k$: all elements){
    ............................
    output $k_{ij}$ of the k-th element
  }
}
```

Inner loop in the GPU code above is actually performed in a single shader with SIMD parallelism.

When an element matrix is symmetric, only the upper triangular matrix or the lower one is needed. This can significantly reduce the number of shaders required to output necessary components of a matrix. To output 300 upper triangular components of the symmetrical stiffness matrix of an 8-noded linear hexahedral solid element, for example, 75 (=300/4) shaders are needed. Each shader, from the shader001 to the shader075, outputs only 4 components of the element matrix as follows:

shader001 outputs $k_{1,1}, k_{1,2}, k_{1,3}, k_{1,4}$,
shader002 outputs $k_{1,5}, k_{1,6}, k_{1,7}, k_{1,8}$,
......
shader074 outputs $k_{21,23}, k_{21,24}, k_{22,22}, k_{22,23}$,
shader075 outputs $k_{22,24}, k_{23,23}, k_{23,24}, k_{24,24}$.

There are two kinds of element mass matrices, consistent mass matrices and lumped mass matrices, and different shaders should be used, respectively. For computing consistent mass matrices, almost the same strategy as that for stiffness matrices, where all the components of the matrix are obtained using one shader after another, is used. The number of components of the consistent mass matrix that should be computed and written, however, can be significantly smaller than that of the stiffness matrix because there are several fixed-to-zero components in the consistent mass matrix that should be omitted from computing/outputting. By utilizing this, the number of shaders required can be significantly reduced. For the consistent mass matrix of an 8-noded linear hexahedral solid element, 48 shaders are enough for computing and writing non-zero components of the matrix in contrast to 75 shaders for the stiffness matrix, and utilizing symmetry further reduces to 27 shaders.

A lumped element mass matrix is a diagonal matrix and has only as many non-zero components as the DOFs per element. So extremely few shaders are enough for computing and writing lumped mass matrices. To output 24 components of the lumped mass matrix of an 8-noded linear hexahedral solid element, for example, only 6 shaders are needed as follows:

shader001 outputs $m^L_{1,1}, m^L_{2,2}, m^L_{3,3}, m^L_{4,4}$,
......
shader006 outputs $m^L_{21,21}, m^L_{22,22}, m^L_{23,23}, m^L_{24,24}$.

The multiple_render_targets (MRT), which enables writing shader output to multiple output textures simultaneously, is available in Open GL Version 2.0 or later [Rost (2006)]. Using both MRT and FBO makes it possible for a shader to write data into four output textures by the latest driver at present. This means that we can reduce the number of shaders to one fourth. Fig. 4 shows schematic illustration of the element matrix computation using MRT. In the case of an 8-noded linear hexahedral solid element, the number of required shaders utilizing symmetry is 19 for computation of stiffness matrices, 7 for computation of consistent mass matrices and 2 for computation of lumped mass matrices.

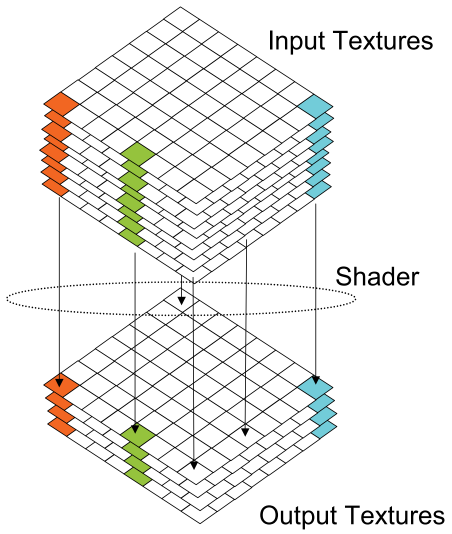


Figure 4: Computation using Multiple Render Targets

3.3 Performance Evaluation

Performance of the GPU computing of element matrices is tested in the following environment:

(1) CPU,Mem.: Pentium4 3.0GHz + 4096MB Memory

(2) GPU: NVIDIA GeForce7800GTX 256MB

(3) OS: Linux Fedora Core 4 + Xwindow(X11R6)

(4) Graphics Driver: NVIDIA 81.78

(5) API: OpenGL 2.0

(6) Shader Language: GLSL

Four meshes which consist of various number of 8-noded linear hexahedral elements are used for performance evaluation. Dividing a cubic domain into cubic elements of the same size makes all meshes. As for material properties, those of steel are used. The total number of elements, that of nodes and that of DOFs of 4 meshes are listed in Table 2.

Table 2: Tested Mesh

	Total Elements	Total Nodes	Total DOFs
Case 1	32768	35937	107811
Case 2	110592	117649	352947
Case 3	262144	274625	823875
Case 4	884736	912673	2738019

Figs. 5, 6, and 7 show the results of performance evaluation for computation of symmetric stiffness matrices, symmetric consistent mass matrices and lumped mass matrices, respectively. In these three figures, the horizontal axis represents the number of elements of the tested mesh, while the vertical axis does the time consumed for computation. The red or solid line with solid circular marks represents GPU computing without MRT, and the solid line with square marks represents the GPU computing using MRT with four output textures. The dashed line with open circular marks that represents the CPU computing is also shown for the purpose of comparison. Equivalent level of code optimization is applied to both CPU and GPU code.

It can be seen from Fig. 5 that the more elements involved, the faster computing element stiffness matrices by GPU with MRT becomes in comparison with that by CPU. Fig. 6 shows similar tendency, but with smaller advantage of GPU because of much less computational intensity in the mass matrix computation compared with stiffness matrix computation. Fig. 7 shows much greater advantage of GPU for lumped mass matrix computation that uses only 7 (without MRT) or 2 (with MRT) shaders. Two lines related to GPU overlap each other in the graph. These results indicate that the more elements, the more intensive computation and the less shaders are involved, the more advantage the GPU shows in comparison with the CPU.

CPU becomes superior to GPU when the total number of elements is small. There are two main causes of this tendency. Firstly, high-speed cache memory of CPU is very efficient for handling small amount of data. Secondly, the high latency of shader execution degrades effective computation speed for a small amount of data/computation. The latter cause also explains the slower increase in GPU computation time compared with CPU for increasing number of elements.

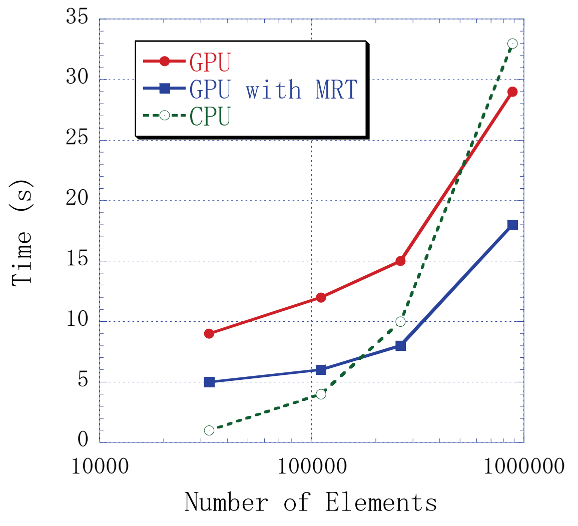


Figure 5: Measured Performance: Stiffness Matrix

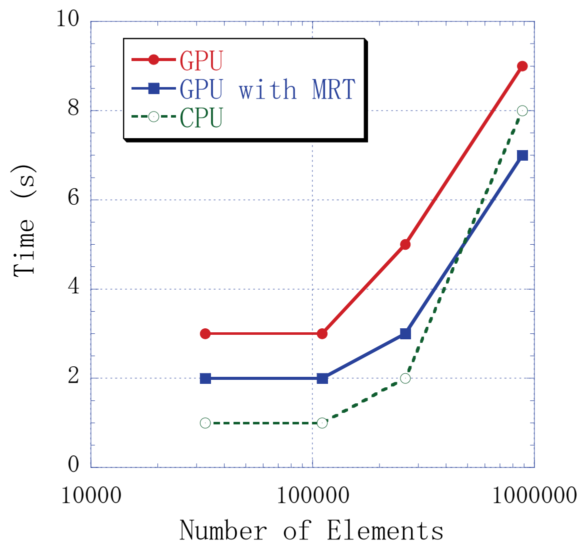


Figure 6: Measured Performance: Consistent Mass Matrix

3.4 Discussions

3.4.1 Textures

There exists a limit of the number of textures accessible from a shader at once. As for Nvidia GeForce6 series or later, for example, this limit

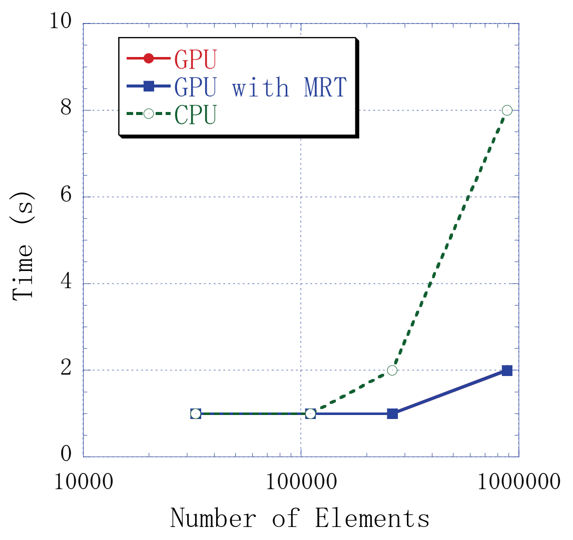


Figure 7: Measured Performance: Lumped Mass Matrix

is no more than 16 that may be insufficient to calculate element stiffness/mass matrices of such elements that consist of tens of nodes or needs many material properties. The limit of the number of accessible textures in these cases can be avoided by using input textures that are several times larger than output textures as shown in Fig. 8. In Fig. 8, the shader fetches data in four contiguous squarely aligned texels in input textures at once and the calculated results are written into the corresponding texel in the output texture.

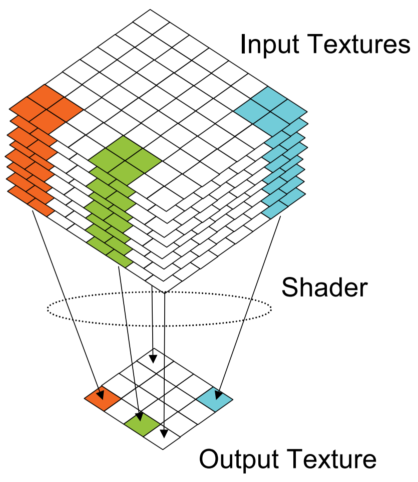


Figure 8: Computation with Reduction

3.4.2 Graphics Memory

Graphics memory on board cannot be so large as CPU main memory; it is usually 256MB - 512MB on today's graphics boards. Considering the construction of element stiffness matrices of 8-noded linear hexahedral solid element, storing 26 32bit floating point values in total, 24 for node coordinates of 8 nodes and 2 material properties, require 104Byte memory per element, and it indicates 104MB for one million elements. This means that the maximum number of elements available at once by the GPU with 256MB graphics memory is, assuming 8-noded linear hexahedral elements and the present implementation, just over one million considering memory consumed for FBO output textures. When several millions or more elements should be computed, they will be divided into several sets of elements less than one million and shaders are to be repeatedly executed to calculate each set of elements in turn.

Another implementation of element matrix calculation on GPUs, where node coordinate data are not previously distributed to texels of corresponding elements, is possible. In this implementation, shown in Fig. 9, coordinate data of all nodes are stored sequentially according to the node number in one texture with just more texels than the total number of nodes, which we call a node texture. Several textures with just more texels than the total number of elements, where each texel represents corresponding element, are also used to store pointers to texels in the node texture, texture coordinates or node number of the nodes in the particular element identified by the texel. While this implementation uses less memory, it causes a lot of random access over the node texture to fetch coordinate data of nodes on execution and may degrade total performance. For our example mesh, this implementation shows performance degradation by almost 10 % for computing stiffness matrices.

3.4.3 Shader Length

There also exists a limit of the number of instructions in a shader. Though this limit for old GPUs was too small to calculate element matrices of FEM, it has become large enough, e.g. 65535 for GeForce6 series or later, for element matrix calculation of elements of basic type. However, it is possible that total instruction count exceeds the limit for higher order or complicated elements. When it happens, we should try to reduce the instruction count of a shader by limiting the number of the output data per shader, limiting the integration points to be calculated per shader, replacing some operations with built-in functions, and so on. Even if instruction count is cut down under the limit, another limit on the temporary register, which is tough to be avoided when it has once arose, may still prevent the shader from working [Lengyel (2003)].

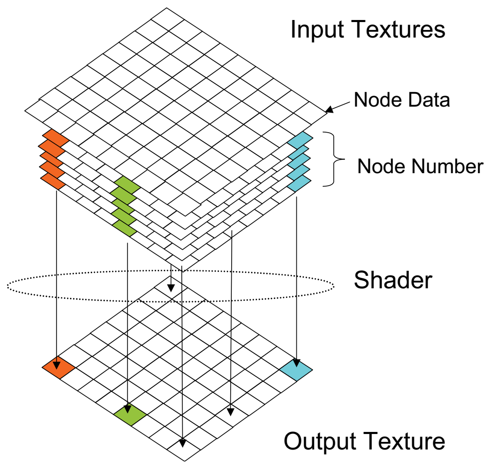


Figure 9: Memory-saving Implementation

4 Time Integration Using GPU

4.1 Time Integration Loop on GPU

Discretization of motion equation of structures of Eq. (2) with respect to time using the central difference scheme leads to the following equation:

$$\frac{1}{(\Delta t)^2}[M]\{u\}_{n+1} = \{f\}_n - \left([K] - \frac{2}{(\Delta t)^2}[M]\right)\{u\}_n - \frac{1}{(\Delta t)^2}[M]\{u\}_{n-1} \quad (8)$$

where $\{u\}_{n+1}$, $\{u\}_n$, $\{u\}_{n-1}$ are displacement vectors at n+1-th, n-th and n-1-th time steps, respectively, and Δt is a time interval. Using the lumped diagonal mass matrix as [M] in the Eq. (8) results in an explicit time integration scheme, which needs no matrix inversion to solve the equation.

In this paper, the explicit time integration scheme utilizing Eq. (8) and a diagonal mass matrix is implemented on the GPU. Eq. (8) can be transformed into the following equation:

$$\{u\}_{n+1} = \{f'\}_n - [K']\{u\}_n - \{u\}_{n-1} \tag{9}$$

where $\{f'\}_n$ and $[K']$ are defined using the identity matrix [I] as follows.

$$\{f'\}_n = (\Delta t)^2 [M]^{-1} \{f\}_n \tag{10}$$

$$[K'] = (\Delta t)^2 [M]^{-1} [K] - 2[I] \tag{11}$$

Multiplying $\{f\}_n$, $[K]$ from left by the inverse of the lumped diagonal mass matrix $[M]^{-1}$ in Eqs. (10), (11) is only a simple row-wise scalar multiplication by the reciprocal of the correspondent diagonal component of $[M]$.

From the right-hand side of Eq. (9), it can be seen that the explicit time integration only consists of two kinds of computations: matrix-vector multiplication, $[K']\{u\}_n$, and addition/subtraction of vectors. We have implemented the time integration scheme, Eq. (9) on the GPU by applying three shaders in turn. The three shaders are applied by the following order:
Shader1: the shader that multiplies the vector$\{u\}_n$ by the matrix $[K']$
Shader2: the shader that performs adding/subtracting vectors$\{f'\}_n - [K']\{u\}_n - \{u\}_{n-1}$
Shader3: the shader that performs copying of vectors $\{u\}_n \rightarrow \{u\}_{n-1}$, $\{u\}_{n+1} \rightarrow \{u\}_n$
Firstly, $[K']\{u\}_n$ is calculated by Shader1. Secondly, the right-hand side of Eq. (9) is calculated by Shader2. This means that time integration calculation of Eq. (9) on the GPU actually needs only two shaders. Finally, copying the vector $\{u\}_n$ to the vector $\{u\}_{n-1}$ and then copying the vector $\{u\}_{n+1}$ to the vector $\{u\}_n$ are performed by the Shader3 to proceed by one time step. A flowchart of this procedure is illustrated in the Fig. 10.

In our implementation, updating the force vector $\{f\}_n$ at each time step is done by firstly updating the corresponding one-dimensional array on the CPU that represents $\{f'\}_n$ in Eq. (10) then loading the array to the texture on the GPU by issuing the corresponding OpenGL call, glTexSubImage2D(). After completing time integration process by one time step, the OpenGL call, glReadPixels(), is called and the $\{u\}_n$ vector is transferred to the correspondent array on the CPU if necessary.

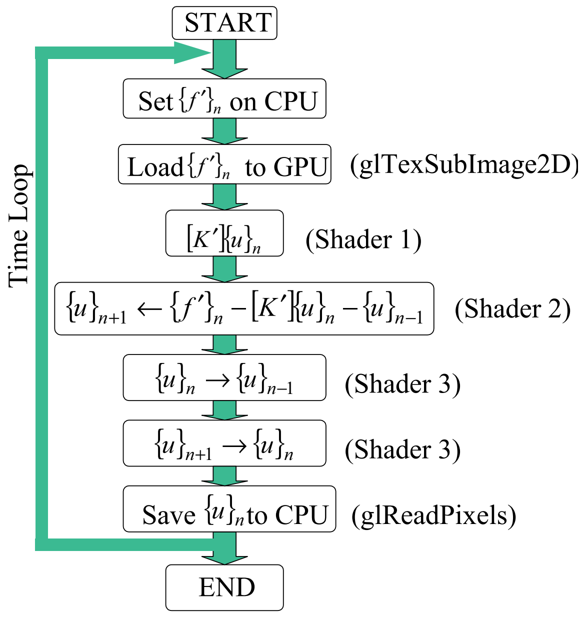


Figure 10: Flowchart of Explicit Time Integration on the GPU

4.2 *Matrix-Vector Multiplication on GPU*

Both addition and/or subtraction of vectors and copying one vector to another can be implemented on the GPU as the operation on the data in a texel of one texture and the data in the texel located at the corresponding coordinates of the other texture. So the shader2 and the shader3, which perform these operations, are easy to achieve high efficiency. Multiplication of a matrix and a vector is, however, ill-suited to GPUs because it is a set of inner-product computations between a row vector of the matrix and the vector, which requires global access over whole components of vectors. In fact, overall performance of the time integration on the GPU depends on the performance of the shader1, i.e. the matrix-vector product shader. An example of a fundamental technique for efficient matrix-vector product is given below.

Consider the following matrix-vector product computation:

$$\vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = A\vec{x} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \tag{12}$$

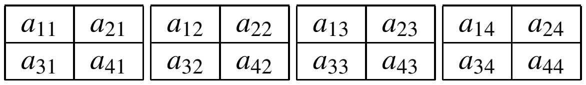
Normal implementation of this equation on the CPU uses doubly nested loops, where scalar product of each row vector of the matrix [A] and the vector $\vec{x}$ is performed in the inner loop. Eq. (13) also shows this order of computation: scalar product computation shown in every dashed box is done in the inner loop one after another.

$$\begin{aligned} b_1 &= \boxed{a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4} \\ b_2 &= \boxed{a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4} \\ b_3 &= \boxed{a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4} \\ b_4 &= \boxed{a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4} \end{aligned} \tag{13}$$

Usual implementation of Eq. (12) on the GPU takes, in contrast, a totally different approach, where all scalar products of every row vector of the matrix [A] and the vector $\vec{x}$ are incrementally and simultaneously calculated. Eq. (14) also shows this approach: each dashed box has a corresponding shader that performs single scalar multiplication and adding the result to the sum at the last step, calculations in these dashed boxes are sequentially done from left to right using the same shader repeatedly.

$$\begin{array}{ccccccccc} b_1 & = & a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & a_{14}x_4 \\ b_2 & = & a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & a_{24}x_4 \\ b_3 & = & a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & + & a_{34}x_4 \\ b_4 & = & a_{41}x_1 & + & a_{42}x_2 & + & a_{43}x_3 & + & a_{44}x_4 \end{array} \tag{14}$$

To realize this approach, the matrix [A] is decomposed into four column vectors and stored as separate textures of 2x2 size, as shown in Fig. 11. One component of vector $\vec{x}$after another is given to the shader as a parameter. Using these textures, calculations in each dashed box, which is performed by one shader, becomes the operation on the data of texels located at the same coordinates of textures, which can be efficiently performed on the GPU.



a_{11}	a_{21}	a_{12}	a_{22}	a_{13}	a_{23}	a_{14}	a_{24}
a_{31}	a_{41}	a_{32}	a_{42}	a_{33}	a_{43}	a_{34}	a_{44}

Figure 11: Matrix Decomposition into Textures

The stiffness matrix of the FEM is usually a sparse matrix in which non-zero components resides only in the neighborhood of its diagonal. Researches have been done for efficient method of matrix-vector product using this type of sparse matrix on the GPU [Bolz, Farmer, Grinspun and Schroder (2003), Krüger and Westermann (2003), Larsen and McAllister (2001)]. Among them, Boltz, Farmer, Grinspun and Schroeder (2003) proposed an approach that uses a texture that sequentially stores only the non-zero components of the original sparse matrix. Krüger and Westermann (2003) proposed an efficient implementation for sparse matrix of banded structure. This approach decomposes the banded sparse matrix into several textures, where each texture stores all data on one diagonally aligned line in the original matrix. This kind of matrix is typical in the finite difference method (FDM). For sparse matrices of this type, this approach is very efficient because the each DOF of the system directly corresponds to the same locations of textures. However, the typical matrix of the FEM with unstructured grid is a banded sparse matrix of different type than that of

FDM: a lot of zero components are usually distributed randomly in the banded area and it can degrade the efficiency of the approach for a strictly banded matrix. In this research, we have implemented a new approach that is based on Krüger and Westermann's approach but improved for FEM matrices. Our strategy of dividing original sparse matrix into several textures is illustrated in Fig. 12. Firstly, non-zero components

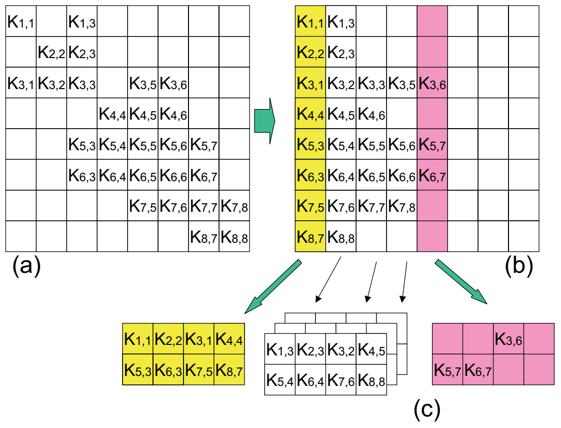


Figure 12: Schematic Diagram of Matrix Storage

in each row are sampled and rearranged from the left end of the row (Fig. 12 (b)). Secondly, each column with one or more non-zero components is stored into textures of which the total number of texels equals to the column size of the original matrix (Fig. 12(c)).

The number of textures required to store all non-zero components of the matrix, which directly decides the number of shaders required that dominates performance, depends on the maximum number of non-zero components per row. If a few rows of the original matrix have much more non-zero components than others, it results in a lot of textures in which most texels only stores zero. This situation is not rare in the FEM with unstructured mesh, and it directly leads to poor computational efficiency. To overcome this, we propose a new technique of sorting rows of the original matrix and performing partial rendering to enhance computational performance by reducing useless calculations with zero. This method is illustrated in Fig. 13 for the same matrix used in Fig. 12 and its procedure is summarized as follows:

(1) All the non-zero components in each row are sampled and rearranged from the left end of the row, as shown in Fig. 13 (a). This process is the same as explained in Fig. 12.

(2) The rows of the matrix are rearranged in an ascending/descending order according the number of non-zero components in each row as shown in Fig. 13 (b). The vector is also rearranged by the same order as rows of the matrix.

(3) Each column vector of the matrix is stored into the texture, as shown in Fig. 13(c). In these textures, non-zero data do not distribute themselves randomly over the whole domain of the texture but resides only in the partial section of the texture because of the rearrangement of rows in procedure (2). This is in contrast to Fig. 12(c), where texels that have zero data and texels that have non-zero data are randomly mixed in the texture.

(4) Computation is executed by rendering the minimum square that covers all non-zero texels. Restricting the rendering area to a small fraction of the textures omits a lot of useless computation otherwise to arise in the texels of zero value, and will enhance performance significantly.

Krüger and Westermann's approach to sparse matrices uses vertex shaders to omit redundant computations with zero components. But, in contrast, our approach does the equivalent only by using fragment shaders combined with reordering the rows of the matrix and reducing the rendering area. The matrix storage format in our approach is equivalent to the Jagged Diagonal Storage [Saad (1989)].

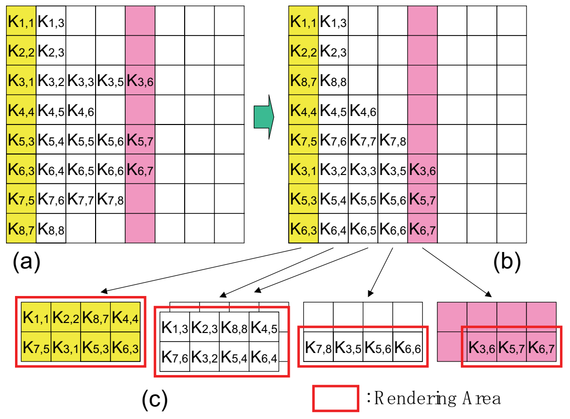


Figure 13: Schematic Diagram of Matrix Storage with row sort

4.3 Performance Evaluation

For performance evaluation, elastic wave propagation in solid is calculated by the dynamic explicit FEM on the GPU and its speed and accuracy are tested. An analysis domain is assumed to be a cubic with all the four corner nodes fixed, and sinusoidal force is applied at the mid area of the upper surface. All elements are 8-noded linear hexahedral solid elements of 1mm x 1mm $\times$ 1mm cube, material properties are assumed to be those of steel, and the time step is set to be 0.01μs. Four kinds of meshes with DOFs varying from 14739 of CASE1 to 823875 of CASE4, which is large enough for practical application, are tested. Specifications of these meshes are listed in Table 3. All tests are done in the environment described in section 3.3.

Table 3: Tested Mesh for Elastic Wave Propagation in solid

	Total Elements	Total Nodes	DOFs
Case1	4096	4913	14739
Case2	32768	35937	107811
Case3	110592	117649	352947
Case4	262144	274625	823875

Fig. 14 shows the speed of elastic wave propagation simulations on the GPU for four CASEs. The vertical axis represents the elapsed time required to proceed by one time step. For comparison, results by CPU, where calculation is executed in double (64bit) precision, are also shown. Though 32bit computation by CPU was also tested, it showed no boost of computation speed. As for the matrix-vector calculation on the GPU, basic implementation, illustrated in Fig. 12, is employed because textures made from the matrix of four CASEs have very few texels of zero value. From Fig. 14, it can be concluded that the GPU, GeForce7800GTX, is almost equal to the CPU, Intel Pentium4 630, in speed of wave propagation simulation.

Fig. 15 shows the accuracy of elastic wave propagation simulation on the GPU for CASE3. The horizontal axis represents time steps, while the vertical axis does the displacement. The thin solid

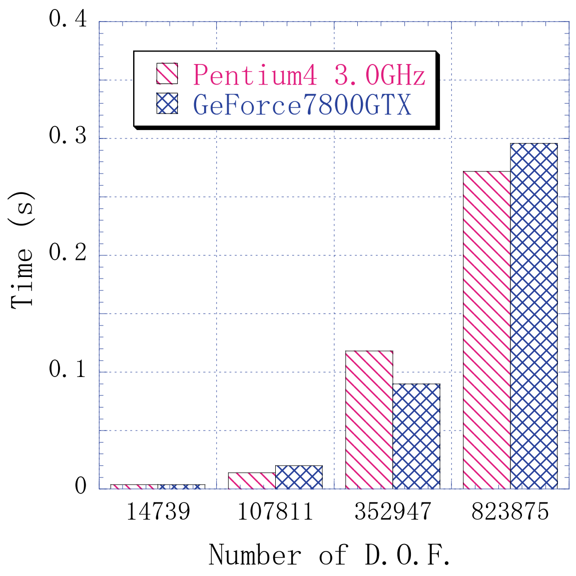


Figure 14: Speed of the GPU-based Wave Propagation Simulation

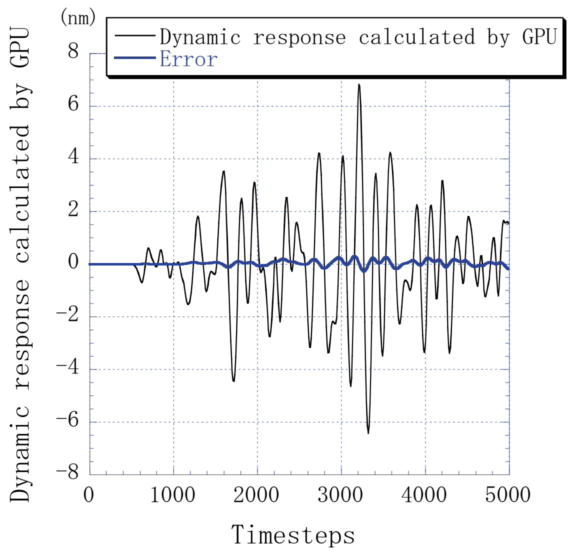


Figure 15: Accuracy of the GPU-based Wave Propagation Simulation

line shows the dynamic response of the displacement in z-direction at the bottom center of the analysis domain calculated on the GPU, and the thick solid line shows the error in the displacement calculated on the GPU compared with that obtained by CPU with double precision. The fact that wave propagation simulations on the GPU do not show any remarkable error and do not indicate any accumulation of error in Fig. 15 proves the feasibility of the GPU in this field.

To assess the effect of the proposed technique of row sort and limited rendering, analyses using CASE4 and CASE5 mesh, listed in Table 4, are tested. The textures derived from the matrix made from the CASE5 mesh are almost equal in total size to those from CASE4 mesh but have much more zero texels. Though these two meshes are regular, comparison between the results using these meshes can show whether the proposed technique works well for unstructured meshes. This is because our target problem uses explicit scheme that is invulnerable to the condition number of the matrix and the performance is expected to depend on the variance of the number of non-zero components per row, which can be assessed by the comparison between CASE4 and CASE5.

Table 4: Tested Mesh

	Total Elements	Total DOFs	Ratio of "0" Elements in Textures
Case4	64×64×64	823875	3%
Case5	2×2×30512	823851	40%

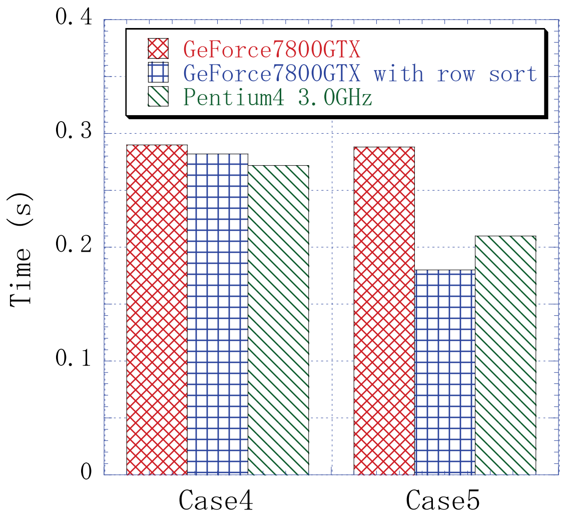


Figure 16: Performance of the Proposed Strategy

Fig. 16 shows the results. The vertical axis represents the elapsed time for simulation of one time step. The elapsed time is measured for three cases: simulation using GPU with row sort and limited rendering area, simulation using GPU without row sort and simulation using CPU for comparison. The elapsed time in the case of GPU without row sort shows almost no difference between CASE4 and CASE5, because the maximum number of non-zero components per row of CASE5 mesh is equal to that of CASE4. This means that almost equal computational workload is required for two cases. In contrast, the elapsed time in the case of GPU with row sort and the case of CPU is significantly shorter for CASE5 than that for CASE4. This is because a number of multiplications by zero are omitted: CPU can avoid useless multiplications by zero simply by omitting them from for-loop scope and GPU with row sort can avoid useless multiplications by zero by omitting them out of the rendering area. Therefore the proposed technique of row sort and limited rendering proved to be efficient for matrices with the varying number of non-zero components per row that usually arise from FEM with unstructured mesh.

5 Discussions

Applicability: Nvidia Corporation, the manufacturer of Geforce series GPUs, has announced a new programming language called CUDA, and CUDA ver.1.0 has been available since June, 2007. CUDA is a high-level language specialized for GPGPU, and it is somewhat similar to the standard C language. AMD Inc., the manufacturer of RADEON series GPUs, has also announced a new programming language called CTM, Close To Metal, which is not publicly available yet. CUDA makes it possible to write GPGPU codes running on GPUs without using graphics APIs, such as OpenGL and DirrectX. This surely makes GPGPU programming for some applications much easier for users. But CUDA only supports Geforce 8x00 GPUs or later, and it does not support any GPUs of AMD and Geforce 7 series or earlier GPUs of Nvidia. This is inconvenient for developing application programs.

In contrast to this, shader code written by using OpenGL and OpenGL shading language is the de facto standard under the present circumstances. Using OpenGL and GLSL as the programming language for GPGPU ensures us that the application code can run on most GPUs irrespective of their manufacturer. Though CUDA and/or CTM will become more and more popular for GPGPU, using a graphics API and a shader language will survive as an alternative because of its broad applicability to various GPUs.

All codes developed in this research are written in OpenGL and GLSL and they can run not only on the GeForce 7800GTX but also on other GPUs. Authors have also tested the wave propagation code of Section 4.2 on three other GPUs and analyzed the CASE5 of Section 4.3. Specifications of the tested GPUs, GeForce 7800GTX, GeForce 7900GTX, GeForce 8800GTX and GeForce 6600 of Nvidia, are summarized in Table 5. They are tested under equal conditions, i.e. the same CPU, the same memory and the same motherboard. These tests were done with the same source code, in other word no modification to the source code was needed.

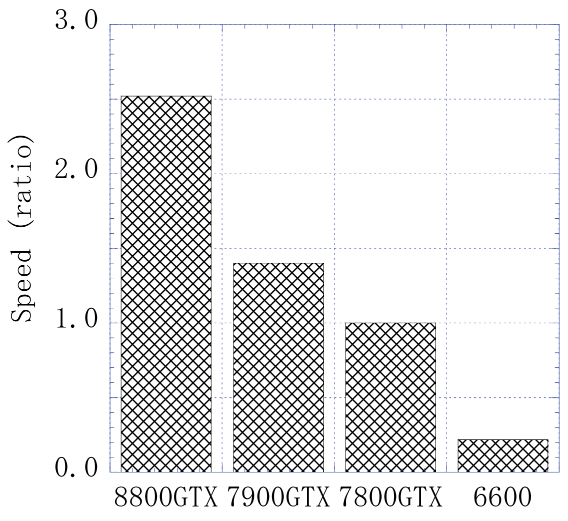


Figure 17: Speed Comparison of several GPUs

Fig.17 shows the measured speed of four GPUs tested. The vertical axis represents the relative speed of each GPU compared to that of GeForce7800GTX, which is normalized to the unit. From Fig.17, it can be expected that the code written in OpenGL and GLSL will work on many GPUs and will be able to perform much better on newer GPUs than the present one without making any modification on the code.

Double Precision: No GPU available today supports double precision. This is a major drawback of today's GPU for CAE applications. Therefore GPUs should only be used for applications that don't require double precision rigorously. There also exists an research to overcome this drawback [Göddeke, Strzodka and Turek (2007)]. In the near future, however, computing in double precision will be supported by GPUs.

Concurrency: Measured performance of GPU doesn't seem overwhelming in comparison to that of CPU. GPU is, however, not for replacing CPU but for helping CPU: GPU can work concurrently with CPU. We can use GPU and CPU as a multiprocessor system suitable for parallel processing. In this context, even GPUs with computational ability comparable to CPU are sufficiently useful as processors in a multiprocessor system consisting of CPUs and GPUs.

Multiple GPUs: Memory on a graphics board is limited in size. This also limits the size of the problem that can be analyzed on a GPU. Parallel processing using multiple GPUs can overcome this limit. We are developing a parallel FEM code based on the domain decomposition technique [Trindade and Pereira (2007), Ha, Seo and Sheen (2006)] for a multiple-GPU system, such as a GPU cluster.

6 Conclusion

GPU is utilized for the finite element analyses of dynamic problems. Two kind of computation are implemented to the GPU, one is the construction of element stiffness/mass matrices and the other is the calculation of time integration based on the explicit scheme. Their basic performance is tested through sample analyses, and the following results are obtained.

1. GPU can calculate a large number of element stiffness matrices twice as fast as CPU.
2. GPU can calculate a large number of lumped mass matrices several times as fast as CPU.

Table 5: Tested GPUs

	Number of Shaders	Clock Rate (MHz)	Memory BandWidth (bit) (MB)	Available since
8800GTX	32	1350	384	Nov. 2006
7900GTX	24	650	256	Mar. 2006
7800GTX	24	430	256	Jun. 2005
6600	8	300	128	Aug. 2004

3. For the explicit time integration scheme in the wave propagation simulation based on FEM, GPU can perform as fast calculation as CPU. With row-sort technique, GPU can perform as fast calculation as CPU even for matrices obtained from unstructured mesh where the number of non-zero components per row in the matrices seriously varies row by row.

These results indicate the feasibility of the GPUs for practical dynamic FEM problems.

References

Astle, D. (ed) (2006): More OpenGL Game Programming. Thomson.

Bathe, K.J. (1996): Finite Element Procedures. Prentice-Hall.

Bolz, J.; Farmer, I.; Grinspun, E.; Schroder, P. (2003): Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, vol.22, pp.917-924.

Boreskov, A. (2006): Developing and Debugging Cross-platform Shaders. A-List. LLC.

Dowd, K. (1993): High Performance Computing. O'Reilly & Associates.

Fernando, R.; Kilgard, M.J. (2003): The Cg Tutorial. Addison-Wesley.

Fernando, R. (ed) (2004): GPU Gems. Addison-Wesley, Boston.

Georgii, J.; Westermann, R. (2005): Mass-spring systems on the GPU. *Simulation Modeling Practice and Theory*, vol.13, pp.693-702.

Göddeke, D.; Strzodka, R.; Turek, S. (2007): Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, vol.22, No.4, pp.221-256.

Ha,T.; Seo, S.; Sheen, D. (2006): Parallel iterative procedures for a computational electromagnetic modeling based on a nonconforming mixed finite element method. *CMES: Computer Modeling in Engineering & Sciences*, vol.14, no.1, pp.57-76.

Harris, M.J.; Baxter III, W.V.; Scheuermann, T.; Lastra, A. (2003): Simulation of Cloud Dynamics on Graphics Hardware. In: Mark, W.A. and Schilling, A. (eds) *Proceedings of Graphics Hardware 2003*. San Diego, Association for Computing Machinery.

Jones, W. (2004): Beginning DirectX9. Premior Press.

Kawai, H. (2004)**:** ADVENTURE AutoGL: A Handy Graphics and GUI Library for researchers and Developers of Numerical Simulations. *CMES: Computer Modeling in Engineering & Sciences*, vol.11, No.3, pp.111-120.

Krüger, J.; Westermann, R. (2003): Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, vol.22, pp.908-916.

Larsen, E.S.; McAllister, D. (2001): Fast Matrix Multiplies using Graphics Hardware, In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC2001)*, Denver, ACM Press.

Lengyel, E. (2003): The OpenGL Extension Guide. Charles River Media.

Li, W.; Wei, X.; Kaufman, A. (2003): Implementing Lattice Boltzmann Computation on Graphics Hardware. *The Visual Computer*, vol.19, pp.444-456.

Müller, M.; Charypar, D.; Gross, M. (2003): Particle-Based Fluid Simulation for Interactive

Applications, In: *Proceedings of ACM SIGGRAPH Symposium on Computer Animation (SCA) 2003*, San Diego, Eurographics Association.

OpenGL Architecture Review Board; Shreiner, D.; Woo, M.; Neider, J.; Davis, T. (eds) (2005): OpenGL Programming Guide 5th Ed. Addison-Wesley.

Owens, J.D.; Luebke, D.; Govindraju, N.; Harris, M.; Krüger, J.; Lefohn, A.E.; Purcell, T.J. (2005): A survey of General-Purpose Computation on Graphics Hardware. In: Chrysanthou and Magnor (eds) *STAR Proceedings of EUROGRAPHICS 2005*, Dublin, Eurographics Association.

Pharr, M.(ed) (2005): GPU Gems2, Addison-Wesley, Upper Saddle River, NJ.

Rost, R.J. (2006): OpenGL Shading Language (Second Edition). Addison-Wesley.

Rumpf, M.; Strzodka, R. (2001): Using graphics cards for quantized FEM computations. In: Hamza, M.H. (ed) *Proceedings of the IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001)*, Marbella, ACTA Press.

Rumpf, M.; Strzodka, R. (2006): Graphics Processor Units: New Prospects for Parallel Computing. *Lecture Notes in Computational Science and Engineering*, vol.51, pp.89-132.

Saad, Y. (1989): Krylov subspace methods on super-computers. *SIAM J. Sci. Stat. Comput.*, vol.10, pp.1200-1232.

St-Laurent, S. (2005): The COMPLETE Effect and HLSL Guide. Paradoxal Press.

Trindade, J.M.F.; Pereira, J.C.F. (2007): On the Efficiency of the Parallel-in-Time Finite Volume Calculation of the Unsteady Navier-Stokes Equations. *CMES: Computer Modeling in Engineering & Sciences*, vol.20, no.1, pp.1-10.

Wu, W.; Heng, P.A. (2004): A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting. *Computer Animation and Virtual Worlds*, vol.15, pp.219-227.