A GPU Implementation of the 2-D Finite-Difference Time-Domain Code using High Level Shader Language

[1] N. Takada, [2] N. Masuda, [2] T. Tanaka, [2] Y. Abe, and [2] T. Ito

[1] Department of Informatics and Media Technology, Sony Institute of Higher Education
Shohoku College, 428 Nurumizu, Atsugi, Kanagawa 243-8501, Japan
ntakada@shohoku.ac.jp

[2] Division of Artificial System Science, Graduate School of Engineering,
Chiba University, 1-33, Yayoi-cho, Inage-ku, Chiba, Chiba 263-8522, Japan

Abstract − The authors have applied a graphics processing unit (GPU) to the finite-difference time-domain (FDTD) method to realize a cost-effective and high-speed computation of an FDTD simulation. The authors used the plane wave scattering by a perfectly conducting rectangular cylinder as the model and investigated the performance of this implementation. The authors timed the computation time of the scattered electromagnetic field by the two-dimensional (2-D) FDTD method at 1,000 steps. Using a PC equipped with an Intel 3.4-GHz Pentium 4 processor and an nVIDIA Geforce 7800 GTX GPU, the authors achieved an approximately 10-fold improvement in computation speed compared with the speed of a conventional central processing unit (CPU) executing the same task.

I. INTRODUCTION

The FDTD method [1] is a numerical technique that can be used to solve electromagnetic boundary value problems in the time domain. This method has excellent numerical accuracy, and is simple to program. Up to now, we have used this technique to solve various electromagnetic field problems such as those pertaining to antennas and electromagnetic scattering [2,3]. However, FDTD simulations for investigating frequency response are computationally expensive. Approaches to this important problem have included modification of the FDTD method and executing the FDTD algorithm on more powerful hardware configurations.

The former approach consists of the alternating direction implicit - FDTD (ADI-FDTD) method [4], and the latter technique consists of a parallel and distributed FDTD method [5–7]. These methods have achieved high-speed computation. However, the ADI-FDTD method is less accurate than the conventional FDTD method, and the parallel and distributed FDTD method requires a supercomputer [7], a PC cluster [5], or a workstation cluster [6], and so is expensive both in financial terms and in the utilization of space.

In recent years, rapid development of powerful GPUs has increased the performance of computer graphics (CG) used for the display of three-dimensional (3-D) images. Current GPUs have a large memory and many programmable graphics pipelines consisting of vertex and fragment processors. For example, the nVIDIA Geforce 7800 GTX has eight vertex and 24 fragment processors with high floating-point performance. We have formulated a program for the GPU using high level shader language (HLSL) and Direct X or OpenGL as a graphics application programming interface (API), called "Shader Program". Vertex and fragment processors can implement looping and floating point math [8,9]. Recently, programmable GPUs have been used for a number of applications other than CG. Traditional physical simulations based on matrix calculations with a GPU have been studied [10-12]. High-speed computer generated holography using a GPU implementation has been reported [13]. From these considerations, it seems that a state of the art GPU would be a cost-effective and very compact device for high-speed computation of FDTD simulation. In the FDTD method, M. J. Inman, et al. reported the GPU code, without absorbing boundaries, written in brook as HLSL and the speedup factors of two different video cards (ATI Radeon 9550 and x800) [14]. The ATI Radeon 9550 and x800 support the 24-bit floating-point format, while the nVIDIA Geforce 6800 GT and 7800 GTX support the 32-bit floating-point format (IEEE 754) [9]. G. S. Baron, et al. coded in OpenGL and used NVIDIA's HLSL, Cg, and discussed speedup and accuracy [15]. Their code included the calculation of the uniaxial perfectly matched layer absorber. However, the Euclidean normalized error increased monotonously with respect to the time steps. However, since they did not investigate the accuracy without absorbing boundaries, the cause of the errors is not confirmed to be the calculation of the absorbing boundary or the 32-bit floating-point format. The present authors believe that the investigation of accuracy without absorbing boundaries is important for the development of the GPU code.

In the present paper, we propose the shader program code to realize accurate and high-speed computation of the FDTD method using a GPU and investigate the basic performance of this computation. We coded in DirectX 9.0c and Microsoft's HLSL because they are well known. When analyzing an electromagnetic boundary value problem using the FDTD method, most of the simulation time is used for the calculation of the electromagnetic fields except at the absorbing boundary. Therefore, we used a simple 2-D model, the plane wave scattering by a perfectly conducting rectangular cylinder, without the absorbing boundary to investigate the basic performance. In the GPU code, the physical parameters are normalized by the electric permittivity ε_0 and magnetic permeability μ_0 in a vacuum space because GPU supports the 32-bit floating-point format. The authors timed the computation time of the scattered electromagnetic field by the FDTD method [3]. The result of the calculation using the GPU only, without the CPU, was approximately a 10-fold improvement in computation speed compared with a conventional CPU (Intel Pentium 4, 3.4-GHz), simulation of the FDTD method. The electric field E_z calculated with the GPU agreed perfectly with that of the CPU in the 32-bit floating-point format. The GPU maintained the accuracy of single-floating point.

The present paper is structured as follows. In Section II, we introduce the 2-D FDTD method. In Section III, we briefly describe a modern graphics hardware device. In Section IV, we describe the implementation of an FDTD simulation using a GPU. In Section V, we detail the performance of the FDTD simulation using the GPU. In the final section, we present conclusions regarding the high-speed FDTD computation using the GPU and describe future research.

II. SCHEME OF THE 2-D FDTD METHOD

In this section, the authors outline the scheme of the FDTD method, which was first proposed by Yee [1]. The basic equations of the 2-D FDTD method in the transverse magnetic (TM) case are as follows,

$$H_x^{n+1/2}(i,j+1/2) = H_x^{n-1/2}(i,j+1/2) - \frac{\Delta t}{\mu \Delta y}\left\{E_z^n(i,j+1) - E_z^n(i,j)\right\}, \tag{1}$$

$$H_y^{n+1/2}(i+1/2,j) = H_y^{n-1/2}(i+1/2,j) + \frac{\Delta t}{\mu \Delta x}\left\{E_z^n(i+1,j) - E_z^n(i,j)\right\}, \tag{2}$$

$$E_z^{n+1}(i,j) = E_z^n(i,j) - \frac{\Delta t}{\varepsilon \Delta y}\left\{H_x^{n+1/2}(i,j+1/2) - H_x^{n+1/2}(i,j-1/2)\right\} + \frac{\Delta t}{\varepsilon \Delta x}\left\{H_y^{n+1/2}(i+1/2,j) - H_y^{n+1/2}(i-1/2,j)\right\}, \tag{3}$$

where $E_z^{n+1}(i,j)$ is the required value E_z of the electric field at the grid point (i, j) and the (n+1)-th time step, Δx and Δy are the sizes of the spatial division in the x and y directions, respectively, and Δt is the time increment. The parameters ε and μ are the electric permittivity and the magnetic permeability in the medium, respectively.

The electric and magnetic fields are evaluated in alternate half-time steps from the initial values with these equations. The FDTD method can finally be used to solve these equations and hence can be used to compute the solution of an electromagnetic boundary value problem in the time domain.

However, in order for the solution to be valid [16], the time increment Δt must satisfy the von Neumann stability condition as follows,

$$\Delta t \le \frac{1}{C_0\sqrt{\Delta x^{-2} + \Delta y^{-2}}} \tag{4}$$

where C_0 is the speed of light in free space.

In the case that a scattering object is a perfect conductor, the scattered electromagnetic fields are as follows,

$$E_z^{scat} = -E_z^{inc} \tag{5}$$

where E_z^{scat} and E_z^{inc} are the scattered electric field and the electric field of the incident wave, respectively.

III. OUTLINE OF MODERN GRAPHICS HARDWARE

In 3-D CG, we model each 3-D object to be drawn on the screen of the host computer in terms of graphics primitives. A primitive is the simplest type of figure: points, lines, triangles, quadrilaterals, and other polygons. Term rendering is used for the process of generating an image on the screen from a model. The GPU has been developed for real-time processing of 3-D CG rendering. Figure 1 shows a block diagram and the dataflow of a conventional graphics hardware device for rendering. The graphics hardware stores the data for rendering, the vertex, texture, pixel data, and the frame buffer, and so on, in the video memory. The frame buffer temporarily stores the image after rendering and is the final target of rendering. The GPU has a pipeline architecture consisting of three parts: the vertex processors, the fragment processors, and the rasterizer. The GPU generally performs the rendering as follows:

(1) The CPU sends the set of vertices of the graphics primitives to the vertex processors.
(2) The vertex processors transform the geometry of the vertices into screen coordinates for display.

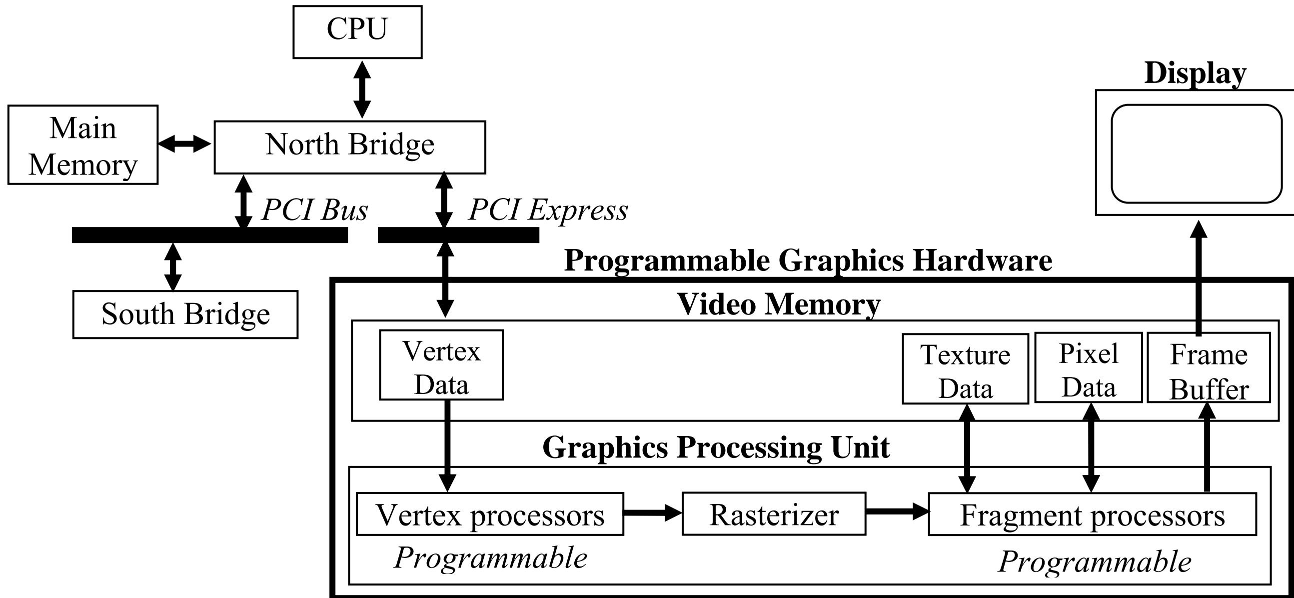


Fig. 1. Block diagram and dataflow of conventional graphics hardware.

(3) The rasterization process is as follows:
(a) The collection of pixels is output by the rasterizer.
(b) The attributes, such as texture coordinates, stored at the vertices are linearly interpolated.
(c) The interpolated value at each pixel is stored.
(4) The fragment processors perform special purpose arithmetic operations on the texture data and compute the resulting final color for each pixel to be drawn on the screen.
(5) The outputs of the fragment processors are sent to the frame buffer in the video memory and the 3-D objects then appear on the display.

With the rapid progress in generating realistic images for computer games have become a requirement for CG to handle large numbers of floating point calculations, resulting in increasingly large and complex rendering implementations. The current GPU has many vertex and fragment processors with high floating point performance. For example, the recently released nVIDIA Geforce 7800 GTX consists of eight vertex and 24 fragment processors, which can perform 32-bit floating-point calculations. The vertex and fragment processors can be utilized as multiple instruction, multiple data (MIMD) and single instruction, multiple data (SIMD) parallel processing units, respectively. Programs to be executed by these processors are written using a shader language and are consequently referred to as shader programs. Programmable GPUs have recently been used for various applications other than graphics. This is known as general-purpose computation on a GPU (GPGPU), of which the present study is an example.

IV. IMPLEMENTATION

We used Microsoft's HLSL as the shader programming language and DirectX 9.0c as the graphics API. Shader programs consist of vertex and pixel shader programs, which are executed on the vertex and fragment processors, respectively. The authors calculate the electromagnetic fields of the FDTD method with the fragment processors in the GPU, because the GPU has more fragment processors than vertex processors. For example, the nVIDIA Geforce 7800 GTX consists of eight vertex processors and 24 fragment processors. In order to calculate the electromagnetic fields with fragment processors, the electromagnetic fields E_z, H_x, and H_y in the computational region of the FDTD method are stored in two textures (temp1_tex, temp2_tex), where "temp1_tex" consists of the electric field E_z in a computational region of the FDTD method and "temp2_tex" consists of the magnetic fields H_x and H_y. If the size of the computational region of the FDTD method is $L_x\ \Delta x \times L_y\ \ \Delta y$, the space division of the x- and y-directions in the textures are $1/L_x$ and $1/L_y$, respectively, because a side of a texture is 1.0.

The vertex and fragment processors in the GPU calculate the electromagnetic fields E_z, H_x, and H_y of the FDTD method by rendering in CG as follows:
(1) Set the vertices (0,0), (1,0), (0,1), and (1,1) on the textures "temp1_tex" and "temp2_tex".
(2) The vertex processors transform the geometry of vertices into screen coordinates for display.
(3) The collection of pixels on textures is output by the rasterizer.

(4) The fragment processors calculate the electromagnetic fields E_z, H_x, and H_y at next time step in parallel.

In the program developed herein, the electric field E_z, the magnetic fields H_x and H_y, and other parameters are stored in GPU registers. The rendering function "VOID Update()" of the CG program written in C++ is shown below.

This program calls the functions for the electromagnetic field (E_z, H_x, H_y) calculation in shader programs. "WIDTH" is the size of the side of the computational region in the FDTD simulation. Here, L_x and L_y are the same. "invTexsize" is the size of a pixel, and "dtdx" and "dtdy" are as follows,

$$dtdx = \Delta t \Delta x \tag{6}$$

$$dtdy = \Delta t \Delta y \tag{7}$$

where Δx and Δy are the space division, Δt is the time increment. "dt" and "times" are the time increment and simulation time, respectively. "BeginPass()" calls each function of the shader programs.

```
VOID Update(LPDIRECT3DDEVICE9 pD3DDev)
{
   for (int step = 0; step < 1000; step++) {
     Hxy->GetSurfaceLevel(0, &pSurf_Hxy);
     pD3DDev->SetRenderTarget(0, pSurf_Hxy);
     pEffect->SetTechnique( hTechnique);
     if (step != 0) {
       pEffect->SetTexture("temp1_tex", Hxy);
       pEffect->SetTexture("temp2_tex", Ez);
     } else {
     pEffect->SetTexture("temp1_tex",initHxy);
       pEffect->SetTexture("temp2_tex", initEz);
   }
     pEffect->SetFloat(hinvTexSize,1.0f/(float)WIDTH);
     pEffect->Begin( NULL, 0 );
     pEffect->BeginPass(0);
     pD3DDev->SetFVF( D3DFVF_CUSTOMVERTEX );
     pD3DDev->SetVertexDeclaration(
                                   pVertexDeclaration);
     pD3DDev->SetStreamSource(0,g_pVB,0,
                               sizeof(CUSTOMVERTEX)
                 );
     pD3DDev-
>DrawPrimitive(D3DPT_TRIANGLESTRIP,0,2 );
     pEffect->EndPass();
     pEffect->End();
     times += dt;
     Ez->GetSurfaceLevel(0, &pSurf_Ez);
     pD3DDev->SetRenderTarget(0, pSurf_Ez);
     pEffect->SetTechnique( hTechnique);
     pEffect->SetTexture("temp1_tex", Hxy);
     if (step != 0) {
        pEffect->SetTexture("temp2_tex", Ez);
     } else {
        pEffect->SetTexture("temp2_tex", initEz);
   }
     pEffect->SetFloat(hinvTexSize,1.0f/(float)WIDTH);
     pEffect->SetFloat(htimes, times);
     pEffect->Begin( NULL, 0 );
     pEffect->BeginPass(1);
     pD3DDev->SetFVF( D3DFVF_CUSTOMVERTEX );
     pD3DDev->SetVertexDeclaration(
                                   pVertexDeclaration);
     pD3DDev->SetStreamSource( 0, g_pVB, 0,
     sizeof(CUSTOMVERTEX) );
     pD3DDev->DrawPrimitive(D3DPT_TRIANGLESTRIP,
                                                0,2);
     pEffect->EndPass();
     pEffect->End();
     pD3DDev->SetRenderTarget(0, pOldBackBuffer);
     pD3DDev->SetDepthStencilSurface(pOldZBuffer);
     pEffect->SetTechnique( hTechnique);
     pEffect->SetTexture("temp1_tex", Ez);
  }
}
```

The pixel shader program is shown below. In this program, "float2" is a 2-D floating-point vector type and "float4" is a four-dimensional floating-point vector type. The parameters from t0 to t3 are input registers of the GPU. "tmep1_samp" and "temp2_samp" are sampler objects for reading "temp1_tex" and "temp2_tex", respectively. The function "PS" returns the values of the electromagnetic fields. The function "PS0" calculates the magnetic fields H_x and H_y (equations (1) and (2)). The function "PS1" calculates the electric field E_z (equation (3)). "VS_OUTPUT0" is the output from the vertex shader. The function "VS0" is the vertex program to transform the geometry of vertices into screen coordinates.

```
float4 PS0 ( VS_OUTPUT0 In ) : COLOR
{
   float hx, hy;
  float ddd;
   float2 t0 = tex2D(temp1_Samp, In.Tex0).xy;
   float t1 = tex2D(temp2_Samp, In.Tex0).x;
   float t2 = tex2D(temp2_Samp, In.Tex0
                  + float2(0.0f, invTexSize)).x;
   float t3 = tex2D(temp2_Samp, In.Tex0
                  + float2(-invTexSize, 0.0f)).x;
   hx = t0.x - dtdy * (t1 - t2);
   hy = t0.y + dtdx * (t1 - t3);
   return float4(hx, hy, 0.0f, 1.0f);
}
VS_OUTPUT0 VS0 (
  float4 Position   : POSITION,
    float2 Texcoord             : TEXCOORD0
){
    VS_OUTPUT0 Out = (VS_OUTPUT0)0;
      Out.Pos = Position;
    Out.Tex0 = Texcoord;
    return Out;
}
```

```
float4 PS1 ( VS_OUTPUT0 In ) : COLOR
{
  float ez;
  float2 t0 = tex2D(temp1_Samp, In.Tex0).xy;
  float2 t1 = tex2D(temp1_Samp, In.Tex0
                + float2(invTexSize, 0.0f)).xy;
  float2 t2 = tex2D(temp1_Samp, In.Tex0
                + float2(0.0f, -invTexSize)).xy;
  float t3 = tex2D(temp2_Samp, In.Tex0).x;
  float2 a;
  float ddd,ams;
  ez = t3 + dtdx * (t1.y - t0.y)
           - dtdy * (t2.x - t0.x);
  a.x=In.Tex0.x * WIDTH;
  a.y=In.Tex0.y * HEIGHT;
  if ( 240 <= a.x  && a.x <= 272 && 240 <= a.y  && a.y <=
                 272) {  /* 1024x1024 */
    ddd=(In.Tex0.x*WIDTH-3.0f)*dx*cos(thetai)
              +(In.Tex0.y*HEIGHT-3.0f)*dy*sin(thetai);
    if(ddd > times){
       ams = 0.0f;
    }else if (ddd > (times - wlamd)){
       ams=(times - ddd)/wlamd * am;
    }else {
       ams=am;
    }
     ez=-ams * sin(omega*(times-ddd));
  }
  return float4(ez, 0.0f, 0.0f, 1.0f);
}

technique FDTDShader
{
 pass P0
   {
     VertexShader = compile vs_3_0 VS0();
     PixelShader  = compile ps_3_0 PS0();
   }
 pass P1
   {
     VertexShader = compile vs_3_0 VS0();
     PixelShader  = compile ps_3_0 PS1();
   }
}
```

The program developed herein is loaded into the GPU and the calculation of the FDTD method is executed by the fragment processor. In this way, the electromagnetic fields were calculated using the FDTD method.

The boundary condition of the perfect conductor (equation (5)) is added in the function “PS1”.

V. PERFORMANCE

The authors used an nVidia Geforce 7800 GTX as the GPU. Table 1 shows the specifications of the Geforce 7800 GTX, which has eight vertex and 24 fragment processors. We timed the calculations required for a simple model to investigate the performance of GPU. As the model, we used the FDTD method to analyze plane wave scattering by a perfectly conducting rectangular cylinder. We used the plane wave as the incident wave, and the electric field E_z^{inc} of incident wave is as follows,

$$E_z^{inc} = E_0 \sin(\omega\, t\Delta t - kx\Delta x) \tag{8}$$

where

$$\omega = 2\pi / 40\Delta t, k = 2\pi / \lambda, \Delta t = 0.5\Delta h / C_0,$$
$$\Delta h = \Delta x = \Delta y = \lambda/20.$$

Table 1. Specifications of the nVidia Geforce 7800 GTX.

Core Clock	430 MHz
Memory	256 MB
Memory Clock	1.2 GHz
Memory Bandwidth	54.4 GB/Sec
Video Memory Interface Width	256 bit
Vertex Shader	8
Pixel Shader	24
API Support	Direct X 9.0c , OpenGL 2.0

The cylinder has an electrical size of kA_s = 10.0, where A_s is the side of the rectangular cylinder. We used equation (5) as the boundary condition on the scattered object. The scattered electromagnetic fields were calculated by the FDTD method.

We compared the GPU system with the CPU system. In the GPU system, we used the FDTD code written in the C++ language and HLSL. All calculations of the FDTD method were performed by only the GPU. For the calculation time of the GPU system, the authors timed 1,000 steps of the calculation using Microsoft Windows XP, Microsoft Visual C++ .NET as the C++ compiler with the options, “-O2” and without threading, and DirectX9.0c as the graphics API. In the CPU system, we used the conventional FDTD code written in the C language, all calculations of the FDTD method were performed by the CPU only, without the GPU. For the calculation time of the CPU system, we timed 1,000 steps of the calculation using two operating systems (OS), Microsoft Windows XP and Linux OS (Fedora Core 4). We used Microsoft Visual C++ .NET as the C compiler with the options “-O2” and without threading. In Linux, we used vmlinuz-2.6.11, not the kernel for Symmetric Multiple Processors, as the kernel and gcc 4.0 as the C compiler with “-O3” as the compiler option.

The specifications of the personal computer used in the GPU and CPU systems were an Intel Pentium 4, 3.4-GHz for the CPU with 2.0 GB of memory.

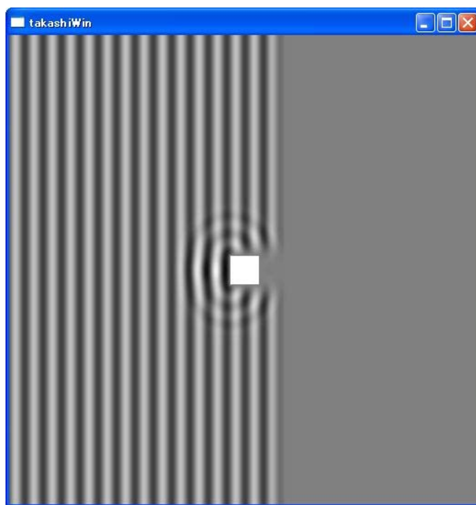
Table 2 shows the calculation time for 1,000 steps for each size of computational region for each system. For a computational region of 1024×1024, the calculation time of the GPU system was 6,340 msec,

while the calculation time of the CPU system using Linux OS: CPU (Linux) was 70,230 msec. Hence, the calculation speed of the GPU system was approximately 11 times faster than that of the CPU (Linux). For the CPU system using Windows XP: CPU (Win), the calculation time of the CPU (Win) system was 74,841 msec. The calculation speed of the GPU system in this case was approximately 12 times faster than that of the CPU (Win). In the GPU system, the computation time is proportional to the number of grid points on computational region of the FDTD method, which means that the fragment processors in the GPU efficiently calculate the electromagnetic field of the FDTD method in parallel. The authors compared the values of the electric field E_z at 1,000 steps for the two systems. The electric field E_z calculated with the GPU agreed perfectly with that of the CPU in 32-bit floating-point format. The GPU maintained single-floating point accuracy.

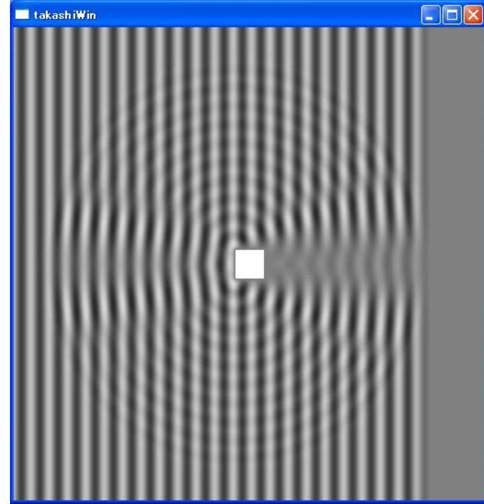
A GPU can directly display the result of FDTD simulation without computation by a CPU. Figure 2 shows the total electric field E_z of the GPU after 600 and 900 steps.

Table 2. Comparison between the calculation time of GPU and that of CPU.

	Computation time of 1,000 steps (ms)		
Size of region	CPU (Win)	CPU (Linux)	GPU(Geforce 800 GTX)
256×256	2047	2138	431
512×512	13763	14020	1655
1024×1024	74841	70230	6340



(a)



(b)

Fig. 2. Total electric field E_z values of the FDTD simulation at two intervals (a) after 600 steps and (b) after 900 steps.

VI. CONCLUSION

The authors developed the program for the calculation of the FDTD method with a GPU and investigated the performance of the GPU by analyzing plane wave scattering by a perfectly conducting rectangular cylinder. In the 1024×1024 computational region, the calculation speed of the GPU (nVidia Geforce 7800 GTX) was approximately 11 times faster than that of a CPU only (Intel Pentium 4, 3.4-GHz). The electric field E_z calculated with the GPU system agreed well with that of the CPU system at 1,000 steps. Finally, we found that the program using HLSL performed high-speed FDTD simulation using the GPU, and the GPU maintained the single-floating point accuracy. Furthermore, the GPU can directly display the result of FDTD simulation without calculation by the CPU. The GPU provides high-speed calculation of the FDTD method and visualization of the electromagnetic field analyzed in the time domain.

In the future, we plan to extend the GPU program by including the code for the implementation of the absorbing boundary and to apply the GPU to the execution of the 3-D version of the FDTD method.

ACKNOWLEDGEMENT

The authors would like to thank Dr. T. Shimobaba and Mr. T. Takizawa for their useful advice.

REFERENCES

[1] K. S. Yee, “Numerical solution of initial boundary value problems involving Maxwell´s equations in isotropic media,” *IEEE Trans. Antennas Propagat*., vol. AP-14, no. 3, pp.302-307, May 1966.

[2] A. Taflove, *Computational electrodynamics: the finite difference time domain method*, Artech House, Inc., 1995.

[3] K. S. Kunz and R. J. Luebbers, *The finite difference time domain method for electromagnetics*, CRC Press, Inc., 1993.

[4] T. Namiki, “A new FDTD algorithm based on alternating direction implicit method,” *IEEE Trans. Microwave Theory Tech*., vol. MTT-47, no. 10, pp.1-5, Oct. 1999.

[5] N. Takada, K. Ando, K. Motojima, T. Ito, and S. Kozaki, “New Distributed implementation of the FDTD method,” *Electronics and Communications in Japan*, *Part 2*, vol. 80, no.5, pp.8-16, 1997.

[6] D. P. Rodohan, S. R. Saunders, and R. J. Glover, “A distributed implementation of the finite difference time domain (FDTD) method,” *Int. J. Numerical Modeling*: *Electronic Networks*, *Devices and Fields*, vol. 8, no.3, pp.283-292, 1995.

[7] D. B. Davidson and R. W. Ziolkowski, “A connection machine (CM-2) implementation of three-dimensional parallel finite difference time domain code for electromagnetic field simulation,” *Int. J. Numerical Modeling*: *Electronic Networks*, *Devices and Fields*, vol. 8, no. 3, pp.221-232, 1995.

[8] nVIDIA corporation, “GPU Gems” Addison-Wisley, 2004.

[9] nVIDIA corporation, “GPU Gems 2” Addison-Wisley, 2005.

[10] J. Boltz, I. Farmer, E. Grinspun, P. Schröder, “Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid,” *ACM SIGGRAPH 03 Proceedings*, 2003.

[11] C. Tompson, S. Hahn, and M. Oskin, “Using modern graphics architectures for general-purpose computing: a framework and analysis,” *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 306-320, Nov. 2002.

[12] J. Krüger and R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms,” *ACM SIGGRAPH 03 Proceedings*, 2003.

[13] N. Masuda, T. Ito, T. Tanaka, A. Shiraki, and T. Sugie, “Computer generated holography using a graphics processing unit,” *Opt. Express*, vol. 14, no. 2, pp.587-592, 2006.

[14] M. J. Inman and A. Z. Elsherbeni, “Programming video cards for computational electromagnetics application,” *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, pp.71-78, Dec. 2005.

[15] G. S. Baron, C. D. Sarris, and E. Fiume, “Fast and accurate time-domain simulations with commodity graphics hardware,” *Proceedings of Antennas and Propagation Society International Symposium*, July 2005.

[16] J. Fang, “Time domain finite difference computation for Maxwell’s equation,” *Ph. D. thesis*, *University of California at Berkley*, 1989.

Naoki Takada Dr. Takada **r**eceived his B.E. and M.S. in electrical engineering from Gunma University, Gunma, Japan in 1994 and 1996, respectively, and his Ph.D. in electrical engineering from Gunma University in 2000. From 1996 to June 2001, he was a research associate at Oyama National College of Technology, Tochigi, Japan. From July 2001 to March 2005, he worked as a research scientist for the High Performance Biocomputing Research Team, Bioinformatics Group, Genomic Science Center (GSC), Institute of Physical and Chemical Research (RIKEN; Yokohama, Japan) and joined the “Protein Explorer Project” for a petaflops special-purpose computer (MDGRAPE-3) system for molecular dynamics simulations of proteins. This project was part of the “Protein 3000 project” supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan. Since April 2005, he has been a lecturer at Sony Institute Higher Education Shohoku College, Atsugi, Japan.

His research interests include GPGPU, distributed and parallel computation including FDTD method, development of a special-purpose computer for FDTD method, numerical simulation including FDTD method, CIP method, and molecular dynamics and electromagnetic theory. He is a member of ACES and IEICE.

Nobuyuki Masuda Dr. Masuda received his BS and MS in System Science from the University of Tokyo (Tokyo, Japan) in 1993 and 1995, respectively, and his Ph.D. in System Science from the University of Tokyo in 1998. From 2000 to March 2004, he was a research associate at Gunma University (Gunma, Japan). Since April 2004, he has been a research associate at Chiba University (Chiba, Japan). Dr. Masuda’s research interests include development of a special-purpose computer for digital holographic particle tracking velocimetry and computer generated holograms on GPU. He is a member of IEICE, IPSJ and ASJ.

Takashi Tanaka Mr. Tanaka received his B.E. in Electronics and Mechanical Engineering from Chiba University (Chiba, Japan) in 2005. He is currently enrolled in the master's program of the Graduate School of Science and Technology, Chiba University (Chiba, Japan). His research interests GPGPU and a high-performance computing of computer generated holograms.

Yukio Abe Mr. Abe received his B.E. and M.S. in electrical engineering from Gunma University (Gunma, Japan) in 1996 and 1998, respectively. In 1998, he began his work at NEC Corporation and engaged in the development of the disk-array system. Since 2006, he has been employed at NEC Co. while pursuing his doctorate at the Graduate School of Science and Technology, Chiba University (Chiba, Japan). His research interests GPGPU and a high-performance computing of a physical simulation.

Tomoyoshi Ito Dr. Ito received B.E., M.S. and Ph.D. from University of Tokyo (Tokyo, Japan) in 1989, 1991 and 1994, respectively. He was a research associate from 1992 to 1994, and an associate professor from 1994 to 1999, at Gunma University (Gunma, Japan). Between 1999 to 2005, Dr. Ito was an associate professor at Chiba University, (Chiba, Japan), and, since 2005, a professor. Dr. Ito's research interests are in high-performance computing and its various applications. He was an initial member of GRAPE project, which has produced special-purpose computers for astrophysics. He developed the first machine, GRAPE-1, in 1989, followed by GRAPE-2 in 1990, among others. Since 1992, he has also designed and built special-purpose computers for the HORN holography system. He is currently investigating three-dimensional television using HORN computers. Dr. Ito is a member of IEICE.