A Fast Runtime Visualization of a GPU-Based 3D-FDTD Electromagnetic Simulation

Kota Aoki, Keisuke Dohi, Yuichiro Shibata, Kiyoshi Oguri, Takafumi Fujimoto

Graduate School of Science and Thechnology, Nagasaki University, Japan
{aoki,dohi}@pca.cis.nagasaki-u.ac.jp
{shibata,oguri}@cis.nagasaki-u.ac.jp
takafumi@nagasaki-u.ac.jp

***Abstract*—In this paper, we present design and implementation of a fast runtime visualizer for a GPU-based 3D-FDTD electromagnetic simulation. We focus on improving the productivity of simulator development without compromising simulation performance. In order to keep the portability, we implemented a visualizer with the MVC model, where simulation kernels and visualization process were completely separated. For high-speed visualization, an interoperability mechanism between OpenGL and CUDA was used in addition to efficient utilization of programmable shaders. We also propose an asynchronous multi-threaded execution with a triple-buffering technique so that developers can concentrate on developing their simulation kernels. As a result of empirical visualization experiments of electromagnetic simulations for practical antenna design, it was revealed that our implementation achieved a rendering throughput of 90 FPS for a viewport of 512 $\times$ 512 pixels, which corresponds to a 12.9 times speedup compared to when the OpenGL-CUDA interoperability mechanism was not utilized. When a standard visualization throughput of 60 FPS was selected, the performance overhead imposed by the visualization process was 15.8 %, which was reasonably low compared to a speedup of the simulation kernel gained by the GPU acceleration.**

I. Introduction

Finite-Difference Time-Domain (FDTD) method is a numerical analysis technique using the Maxwell's equation in spatial and time domain and calculates the electric and magnetic potential at each point on spatial grids or lattices[1][2]. Although huge computational costs and memory usages are required for practical size of simulation models, the method is widely used since performance of computers has been rapidly improved and the algorithm is simple and easy to understand. Many researchers proposed various implementation and optimization techniques for acceleration platforms of the FDTD method[3]. We also implemented a 3D-FDTD electromagnetic field simulation on GPU and proposed novel GPU-oriented boundary conditions[4]. While GPU acceleration was shown to be effective for the algorithm, it was also revealed that efficient programming for GPUs often requires a deep understanding of architectures and parallel processing. It is heavily restricting the productivity of application development for GPGPU[5][6].

In this paper, we present design and implementation of an efficient runtime visualization framework for GPGPU scientific simulations, aiming at increasing the productivity of application development on GPUs. With this visualizer, developers will be able to easily understand behavior of the entire simulation space and detect bugs in an early stage of development.

In order to achieve real-time visualization of GPU simulation results, data transfer between a host and the GPU needs to be carefully reduced. A naive operation flow may first launches a GPU kernel and transfers the results calculated by the kernel from the GPU to the host. Then, the results are converted to color information on the host CPU. Finally, the color information is sent back again to video RAM (VRAM) in the GPU and displayed. Iterating these three processes, simulation behavior is visually displayed. However, data transfer via PCI-Express occurs twice for a single simulation step, severely degrading the simulation performance.

This problem can be solved by using OpenGL interoperability APIs offered by CUDA, which allows us to directly store execution results of GPU kernels to the VRAM on the GPU, without ping-ponging the data on the PCI-Express. However, programming code for simulation process and visualization process are tightly unified in this approach. This means that developing a new simulation program also requires rewrite of visualization code to be adjusted for the new GPU kernel, resulting in further deterioration of the productivity of application development.

Therefore, we designed our framework based on a software architecture pattern called the model-viewer-controller (MVC) model. In this model, simulation code and visualization code are definitely separated, to keep the portability. Multi-threading is also introduced to enable concurrent processing between the simulation kernel and the visualization process. These performance-oriented techniques are hidden from an application level; developers can easily visualize their simulation results, without being aware of the graphics APIs and without affecting the simulation performance.

The rest of the paper is organized as follows. Section II introduces related works and Section III describes the 3D-FDTD method and implementation of 3D-FDTD method on a GPU. Section IV describes the purpose and design policy of the proposed framework. Section V shows implementation details. Performance evaluation results are presented in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Various visualization techniques for GPGPU have been proposed so far. In[7], using OpenGL interoperability APIs offered by CUDA a bottleneck of visualization process for N-body simulations on a GPU was eliminated. As a result, a rendering speed of 30.1 frame per second (FPS) was achieved for 2,744 particles. In[8], GPU based algorithms for solving and visualizing level-set computation with real-time volume rendering was proposed. This method was shown to be 10 to 15 times faster than CPU-based implementation. A large-scale volume rendering system for a PC cluster equipped with GPUs has also been proposed[9]. Using a 16-node PC cluster, a rendering speed of 5 FPS was achieved for 2048 $\times$ 1024 $\times$ 1878 drawing data.

More flexible visualization frameworks have also been proposed to improve visualization performance for specific applications. In[10], a framework for GPU-based volume rendering that offers extensibility in terms of shader functionality has been proposed. Users can easily modify or replace the shader functionality on this framework. The Zippy is a scalable high-performance computing framework with visualization for a GPU cluster[11]. This framework was proposed to solve complexity problem of a programming model for high performance general purpose computation on GPU clusters. The Zippy provides facilities for parallel programming and debugging as well as a generalized visualization. In[12], an interactive out-of-core technique was proposed and implemented as a framework of a single-pass GPU ray-casting for rendering massive scalar volumetric data sets.

However, as far as our knowledge goes, there have not been any work on visualization frameworks that introduced multi-threading and CUDA-OpenGL interoperability, focusing on simultaneous realization of high-speed simulation and general versatility.

III. FDTD

A. 3D-FDTD Method

Finite-difference time-domain (FDTD) method for electromagnetic simulation proposed by Yee discretizes the Maxwell's equation in spatial and time domain and calculates behaviors of electric and magnetic fields at each point on spatial grids[1][2]. In this implementation, we assumed that a simulation target is isotropy and non-dispersive having the conductivity of $\sigma = 0$. Let $\boldsymbol{E}$, $\boldsymbol{H}$ and μ denote an electric field, a magnetic field, and permeability, respectively. This yields Maxwell's curl equations:

$$\begin{aligned} \frac{\partial \boldsymbol{E}}{\partial t} &= \frac{1}{\varepsilon} \nabla \times \boldsymbol{H} \\ \frac{\partial \boldsymbol{H}}{\partial t} &= -\frac{1}{\mu} \nabla \times \boldsymbol{E} \end{aligned} \tag{1}$$

Considering a typical substitution of central differences for the time and space derivatives, we got the following time-stepping expression of Maxwell's curl equations:

$$\begin{aligned} E_x^n\left(i+\frac{1}{2}, j, k\right) &= E_x^{n-1}\left(i+\frac{1}{2}, j, k\right) \\ &+ \frac{\Delta t}{\varepsilon \Delta y} \delta_y H_z^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j+\frac{1}{2}, k\right) \\ &- \frac{\Delta t}{\varepsilon \Delta z} \delta_z H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j, k+\frac{1}{2}\right) \end{aligned} \tag{2}$$

where

$$\delta_y H_z^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j+\frac{1}{2}, k\right) =$$
$$H_z^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j+\frac{1}{2}, k\right) - H_z^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j-\frac{1}{2}, k\right)$$

$$\delta_z H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j, k+\frac{1}{2}\right) =$$
$$H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j, k+\frac{1}{2}\right) - H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2}, j, k-\frac{1}{2}\right).$$

By taking a difference in the space domain, $1/2$ offsets arise in some dimensions for each component of $\boldsymbol{E}$ and $\boldsymbol{H}$. Similarly, due to taking a difference in a time domain, $\boldsymbol{E}$ and $\boldsymbol{H}$ are stored in different time-steps. While $\boldsymbol{E}$ is aligned at time-step n, $\boldsymbol{H}$ is aligned at time-step $(n-1/2)$, where n is an integer value. E_x^n shows the component x of the electric filed $\boldsymbol{E}$ at time steps n. $H_y^{n-(1/2)}$ and $H_z^{n-(1/2)}$ denote components y and z of the magnetic field $\boldsymbol{H}$ at time-step $(n-1/2)$, respectively. The time increment in the simulation is denoted as Δt and permittivity for each space grid point is expressed as ε. Δy and Δz are the grid size in the y and z coordinate directions. We use time-stepping expressions for E_y^n, E_z^n, $H_x^{n-\frac{1}{2}}$, $H_y^{n-\frac{1}{2}}$, $H_z^{n-\frac{1}{2}}$ in the same manner. Simulations of electromagnetic propagation are performed by iteratively calculating these time-stepping expressions.

B. Implementation of 3D-FDTD on a GPU

Like a common approach of GPU implementation of stencil computation[3][13], our implementation also divides the whole simulation space of size (S_x, S_y, S_z) into small *blocks* of size (B_x, B_y, B_z) and makes CUDA thread blocks process each *block*. Data for $\boldsymbol{E}$ and $\boldsymbol{H}$ fields are stored in memory as 3D-array so that dimension x has a unit-stride, dimension y has a larger stride, and dimension z has the largest stride. In Figure 1, the placement and processing direction of CUDA threads within a *block* are shown. $B_x \times B_y$ CUDA threads are placed on a 2D plane and each CUDA thread goes along the line with z direction. That is, the CUDA thread on the coordinate (x, y) calculates a total of B_z cells from $(x, y, 0)$ to $(x, y, B_z - 1)$. As an absorbing boundary condition, a modified version of split perfect matched layers (PML), which we proposed for efficient GPU implantation, is utilized[4].

IV. Design Policy

In order to achieve a high degree of portability and expandability of visualization process, our framework was designed

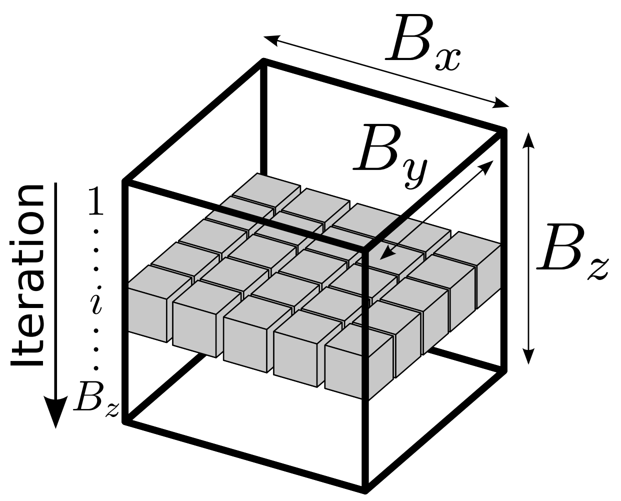

Fig. 1. Placement of CUDA threads and direction of process within a *block*. The gray boxes show CUDA threads.

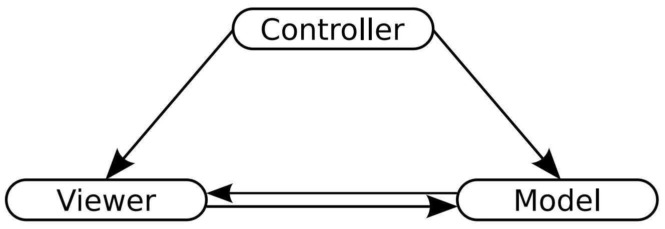

Fig. 2. MVC model

with a software architecture pattern called the model-viewer-controller (MVC) model. Figure 2 shows an overview of the MVC model, which consists of three independent software elements called "Model", "Viewer", and "Controller". The "Model" module performs GPU kernel functions for simulation which are implemented by application developers. The "Viewer" module renders execution results calculated by the "Model". The "Controller" module handles the entire behavior of the simulator. Our framework offers the latter two modules. For the "Viewer" part, volume rendering process has been implemented with programmable shaders using OpenGL Shading Language (GLSL)[14] at this time. Since the "Model" and the "Viewer" parts are completely separated in the MVC model, application developers can implement their simulation kernels and visualize simulation results without being aware of detailed graphics APIs such as OpenGL.

As discussed in Section I, our framework supports the interoperability mechanism with OpenGL offered by CUDA to remove the bottleneck caused by heavy data transfer between a host and a GPU. In OpenGL, a memory region on a GPU called Buffer Object stores rendering data such as pixels and vertices. The OpenGL interoperability APIs offered by CUDA bind Buffer Objects and CUDA memory regions so that execution results calculated by GPU kernels are directly stored into the Buffer Object which is registered in advance. In this way, the data transfer via PCI-Express is effectively reduced.

A multi-threaded control flow is another important feature of our framework. Since the "Viewer" module needs to handle asynchronous inputs from users such as mouse operations and window exposures, common OpenGL applications have a simple control flow consisting of a single event loop. However, when CUDA kernels are involved in such an event loop, the "Viewer" and the "Model" mutually interfere each other's performance, resulting in a slowdown of the simulation. To cope with this problem, we introduced a multi-threaded control flow, which allows the "Viewer" and the "Model" to be concurrently executed while being described under independent control flows. This approach is advantageous in terms of both performance and productivity.

However, since some of OpenGL functions are not thread-safe, the "Model" is not able to write to the Buffer Object that is being rendered by the "Viewer". An intuitive solution to this problem would be a double buffering technique illustrated in Figure 3, where two Buffer Objects are provided. When the "Model" is storing results to one Buffer Object, the "Viewer" reads out the contents of the other Buffer Object. Then, after finishing this process, the buffers are swapped. Since this swap operation can be issued only by the "Viewer" in OpenGL, the "Viewer" and the "Model" need to be synchronized. This raises the problem of mutual performance interference again.

Therefore, we employed a triple buffering approach as shown in Figure 4. Having another additional Buffer Object, there is always one buffer that is not accessed, so that both the "Viewer" and the "Model" can switch buffers any time without synchronizing each other. Still there is a possibility that the "Viewer" and the "Model" switch to the same buffer at the same time by chance, but simultaneous access to the same Buffer Object can be prevented by a mutual exclusion control mechanism. When the execution speed of the "Model" is faster than that of the "Viewer", frequent buffer switching by the "Model" causes an overwrite of the Buffer Object that has not been rendered by the "Viewer" yet. However, some dropping frames will be acceptable if the "Viewer" provides a sufficient rendering throughput such as 60 FPS. On the other hand, when the execution speed of the "Model" is slower than that of the "Viewer", the "Viewer" simply continues to render the same Buffer Object.

V. Implementation

A. Overview of the Framework

As illustrated in Figure 5, we implemented the framework with four units: "Controller", "Model", "Viewer" and "Data Storage". Where application developers implement their own simulation kernels is the "Model". Inheriting and expanding the template, developers can easily implement simulation kernels without being aware of details of graphics APIs. The "Viewer", which was implemented by OpenGL APIs, renders execution results calculated by the "Model". Programmable shader using GLSL was introduced in order to realize high-speed and flexible volume rendering.

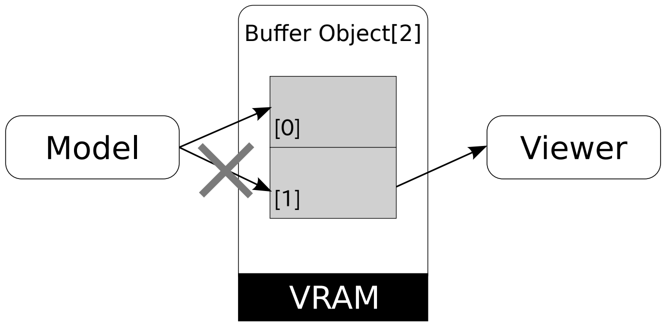

Fig. 3. Double buffering approach

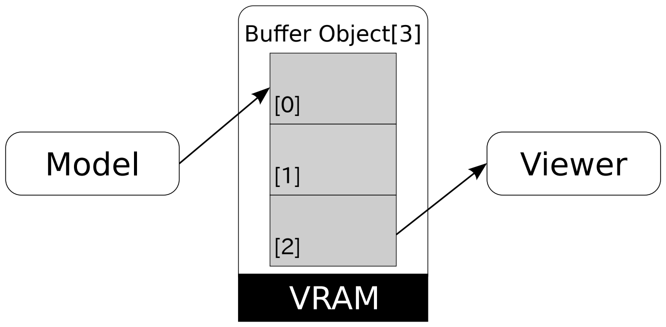

Fig. 4. Triple buffering approach

The "Data Storage" is a module that holds and manages drawing data which calculated by the "Model" with the triple buffering approach described in the previous section. Using the interoperability mechanism with OpenGL offered by CUDA, efficient data transfer from the "Model" to the "Viewer" was implemented. The "Controller" manages the entire behavior of the framework and initializes of the other modules. The multi-threaded execution model was introduced in order to provide a view of a simple control flow to developers and to minimize interference between the "Model" and the "Viewer" in terms of performance. The mutual exclusion facilities required for the "Data Storage" were implemented using the Boost C++ library.

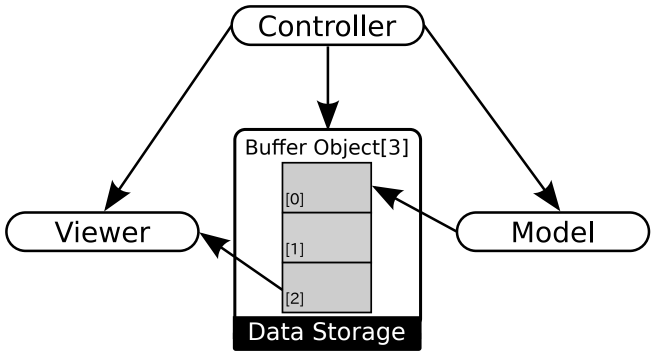

Fig. 5. Overview of the framework

B. Data Storage

In the "Data Storage" module, three buffer regions for drawing data to perform the triple buffering are allocated when this module is initialized by the "Controller". This module always monitors which buffer is used by which module as well as which buffer has the latest simulation results that have not been rendered yet. When the "Model" or the "Viewer" requests buffer switching, the "Data Storage" provides an appropriate buffer region. Thus, the "Data Storage" effectively absorbs the speed gap between the "Viewer" and the "Model" preventing mutual performance interference.

When the simulation kernel in the "Model" is implemented in CUDA, the "Data Storage" also manages the data transfer using the interoperability APIs between OpenGL and CUDA. The detailed mechanisms such as the triple buffering and OpenGL-CUDA interoperability are hidden from developers of the "Model". The "Model" gets a pointer to the appropriate buffer from the "Data Storage" every time the buffers are switched. What the "Model" needs to do is to write the simulation results to the region the pointer indicates and to request buffer switching.

C. Controller

The "Controller" is provided as a simple template-code which developers can be easily modified if they need. Figure 6 shows example code of "Controller" which has a bare minimum of works that "Controller" needs to do.

After instantiating the "Model" object (Line 6), the buffers to store drawing data are allocated and initialized in "Data Storage" (Line 8 to 12). Here, the "Controller" checks whether the interoperability mechanism between CUDA and OpenGL is used or not, and an appropriate type of buffer structure is allocated by invoking the corresponding function.

Then, "Viewer" is initialized (Line 16) and executed as another thread (Line 17). After the "Model" is initialized and the same instance of "Data Storage" is shared (Line 19), the simulation kernel in the "Model" is invoked from the simulation loop constantly (Line 21 to 23).

D. Model

Figure 7 shows the definition of the basic class of "Model", which is to be inherited and expanded by developers. As the code shows, developers do not need to take care of OpenGL APIs to visualize simulation results and can concentrate on their simulation kernel, which is a major advantage in terms of the productivity.

The actual simulation kernel is implemented into Process() function which is shown in Figure 7. In this function, a method provided by "Data Storage" is invoked in order to get a pointer to an appropriate Buffer Object which simulation results are written into. This pointer has the same as pointers the cudaMalloc() function returns and the corresponding Buffer Objects are allocated on memory of the GPU. Meanwhile, a pointer to the "Data Storage" object is passed when the "Controller" calls initialization function.

```
int main() {
  IViewer *p_viewer = NULL;
  IModel *p_model = NULL;
  IVolumeDataFactory *p_data_factory =
      NULL;

  p_model = new fdtd::ModelFDTD();

  if(p_model->IsCUDAGlBridgeEnable()) {
    p_data_factory = new
        VolumeDataCUDAFactory();
  } else {
    p_data_factory = new
        VolumeDataCPUFactory();
  }

  p_viewer = new ViewerGl(p_data_factory);

  p_viewer->Init();
  p_viewer->Invoke();

  p_model->Init(p_viewer->
      GetCurrentStorage());

  while(!p_model->IsQuit()) {
      p_model->Process();
  }

  delete p_model;
  delete p_viewer;

  return 0;
}
```

Fig. 6. Sample code of Controller

The IsCUDAGlBridgeEnable() function can control the interoperability mechanism between OpenGL and CUDA to be used or not, depending on a Boolean return value. Thus, the framework can be used to visualize simulation kernels that are not implemented in CUDA.

Developers should write the termination conditions into the IsQuit() function. When the simulation terminated, this function returns true. "Controller" received return value, and after that, visualizing simulation stopped.

E. Viewer

"Viewer" is a module where simulation results are rendered and was implemented with OpenGL and GLUT libraries. Multi-threading execution was carried out by introducing the Boost::thread class in Boost C++ Libraries. Figure 8 shows a display function of "Viewer". This function is called from glutDisplayFunc() which is registered in glutMainLoop(). The pointer to the Buffer Object that should be rendered is passed from "Data Storage". Using GetLatestViewerData(), the latest simulation results are obtained and displayed. Each voxel has four data elements corresponding to RGBA. A type of drawing data can be chosen from three options: "float", "int" and "unsigned int". In this implementation, the size of each

```
class IModel {
 public:
  IModel() {}

  virtual void Init(DataStorage *storage)
      =0;

  virtual void Process()= 0;

  virtual void Quit() = 0;

  virtual bool IsQuit() = 0;

  virtual const std::string &GetName()
      const = 0;

  virtual const std::string &
      GetArchitecture() const = 0;

  virtual bool IsCUDAGlBridgeEnable() = 0;

  virtual IModel() {}
}
```

Fig. 7. IModel.hpp

dimension "x", "y" and "z" of the drawing space is confined to power-of-two numbers.

"Viewer" calls glutPostRedisplay() in glutIdleFunc() to redisplay the simulation results calculated by "Model". To control the visualization throughput, sleep() was inserted before calling glutPostRedisplay(). The default target throughput is 60 FPS, while the value can be easily changed.

At present, the "Viewer" supports volume rendering, which is a technique to generate 3-D images from brightnesses and transparency rates of volume data. By controlling the transparency rates, it is possible to display an specified cross-section surface of a given target object and is also possible to make the object translucent to effectively express internal structure. Thus, the technique is widely used to visualize computing results of scientific simulations.

Figure 9 shows an overview of the volume rendering technique. For each pixel in the image plane, the voxel data that cross with the lines of sight are summed up using the ray casting algorithm. When the accumulated permeation rate (alpha value) reaches a threshold defined in advance or the ray falls out of the drawing space, the intensity of the corresponding pixel on the image plane is calculated from the accumulated voxel data at that point. Although the method was not suited for runtime and interactive visualization due to its high computing complexity in past days, high speed rendering is now feasible using GPUs technologies[15].

In our implementation, vertex shaders and fragment shaders on the GPU were utilized to implement high-speed ray casting. While the vertex shaders set up insight vectors, the fragment shaders are responsible for calculation of the voxel data. The threshold alpha value of the accumulation was set to 0.95.

```
void ViewerGl::Pimpl::GLUTDisplay() {
  gl::VolumeData *p_volume_data =
      m_data_storage->GetLatestViewerData
      ();
    ...
    m_frame_buffer->Bind();
    glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);

    static gl::Cube cube;
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    cube.Draw();
    glDisable(GL_CULL_FACE);

    m_frame_buffer->Unbind();
    glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);

    glUseProgram(m_shader->
        GetShaderProgram());
    glUniform1i(m_shader->
        GetUniformLocation("RayEnd"), 0);
    glUniform1i(m_shader->
        GetUniformLocation("VolumeData"),
         1);
    glUniform3f(m_shader->
        GetUniformLocation("TextureSize")
        , p_volume_data->GetSizeX(),
        p_volume_data->GetSizeY(),
        p_volume_data->GetSizeZ());

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D,
        m_frame_buffer->GetColorBuffer())
        ;
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_BUFFER,
        p_volume_data->GetTexture());

    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    cube.Draw();
    glDisable(GL_CULL_FACE);

    glUseProgram(0);
    glutSwapBuffers();
  }
```

Fig. 8. GLUTDisplay function

VI. Evaluation and Discussion

A. Conditions for Evaluation

To evaluate the proposed framework, we used 3D finite-difference time-domain (FDTD) electromagnetic simulation kernel[4] as a simulation model. In this electromagnetic simulation, characteristics of a microstrip antenna for vehicles are obtained. The simulated grid size is (224, 212, 96) and single precision floating point arithmetic is used. The sizes of a drawing space and a viewport for the volume rendering were set to (256, 256, 256) and (512, 512), respectively. The simulation time step was set to 15,000, which is enough to calculate characteristics of the target antenna.

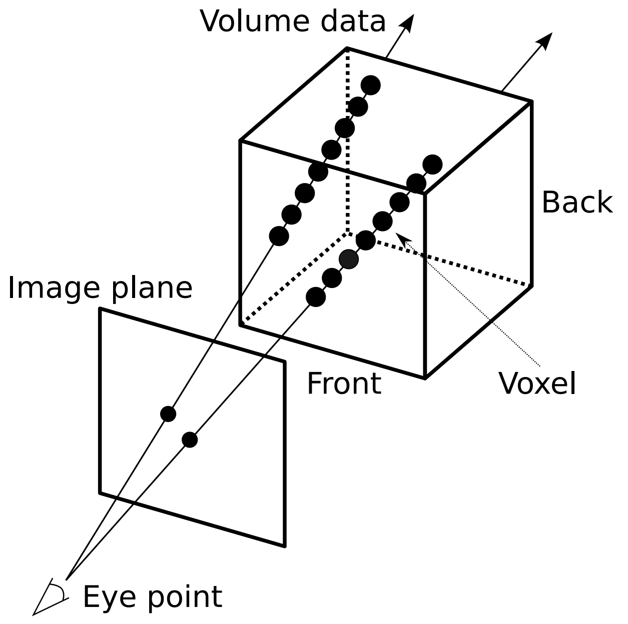

Fig. 9. Overview of volume rendering

The implementations are carried out on a GeForce GTX 580, which has GF110 GPU core architecture with 512 CUDA cores and 3-GB GDDR5 memory. We used 3.40-GHz Core i7-2600K CPU with 16-GB DDRIII-1066 memory as the host PC. The proposed framework was developed and executed in software environment with CUDA 4.2, OpenGL 4.2.0, g++ 4.6.2., and OpenSUSE 12.1. Figure 10 shows an example screenshot of execution results for the evaluated simulator.

B. Evaluation of the Interoperability Mechanism

In order to evaluate the effect of the interoperability mechanism between CUDA and OpenGL on visualization performance, we implemented the same framework without the interoperability APIs and compared the performance. Figure 11 shows relationship between target FPS and measured FPS for both of the implementations. Here, the target FPS means a visualization speed set by the user and the measured FPS shows the visualization speed that was actually achieved by the framework. When we did not utilize the interoperability mechanism for CUDA and OpenGL, measured FPS was saturated at around 7.0 FPS. Meanwhile, the highest measured FPS of 90 FPS was achieved by using the interoperability APIs. This corresponds to a 12.9 times speedup.

C. Impact of Visualization Process on Simulation Performance

Next, we evaluated how the visualization process affected the execution time of the simulation by changing the target FPS. Figure 12 shows the measured results. In this figure, the execution time of the simulator without the visualization is also illustrated as “Without visualization”. Without the interoperability APIs, a severe slowdown was shown when the target FPS exceeded 5 FPS. Meanwhile, the simulation performance was not largely affected by the target FPS when the interoperability APIs were utilized. Interestingly, unlike the visualization performance in the case of Figure 11, we did not find any radical changes in the simulation performance around 90 FPS. These results suggest that our multi-threading approach was successfully introduced to prevent the simulation performance from being interfered by the visualization performance.

D. *Effect of Visualization Process*

Compared to the performance of the simulation without the visualization process, the simulation time on our framework was degraded by 15.8 % for 60 FPS, which is a common visualization speed most people feel smooth. The main reason of this overhead is that computing resources on the GPU are allocated not only to the simulation kernel but also to the visualization process. Actually, the overhead was slightly increased as the target FPS increases as shown in Figure 12. The effect of the interoperability APIs were remarkable also for the simulation performance, showing an approximately 6 times speedup.

E. *Effect of GPU Acceleration*

We also compared the performance of the 3D-FDTD simulation with a CPU-based implementation, in order to reveal the effect of GPU acceleration. The same simulation algorithm was implemented on the PC system described in Section VI-A without using the GPU. While parallel processing with multi-core, multi-thread and SIMD instructions were not used in this implementation, the code was optimized considering the cache structure. Figure 13 shows the comparison results. Even with the use of visualization facilities, still 28.8 times speedup was obtained by the GPU acceleration. In this context, we think the visualization overhead of 15.8 % is reasonably acceptable in compensation for increased productivity of application development.

F. *Memory Usage*

Our framework introduced the triple-buffering technique which needs a relatively large memory space on the GPU. The total memory requirement M_{Total} for the triple-buffering can be shown as:

$$M_{\text{Total}} = N_v \times N_c \times S_t \times N_b$$

where N_v, N_c, S_t and N_b are the number of voxels in the drawing space, the number of color channels for each voxel, the size of drawing data type in bytes, the number of buffers, respectively. In this implementation, $N_v = 256^3$ and $N_c = 4$ since the RGBA color channel format was used. In this implementation, $S_t = 4$ since the single precision floating point arithmetic was used and $N_b = 3$ because of the triple-buffering, the total required size is 768 MB. On the other hand, the size of device memory on the GPU required by the FDTD kernel for this experimentation was 205.1 MB. Since the evaluated GPU has a 3 GB GDDR5 memory capacity, the

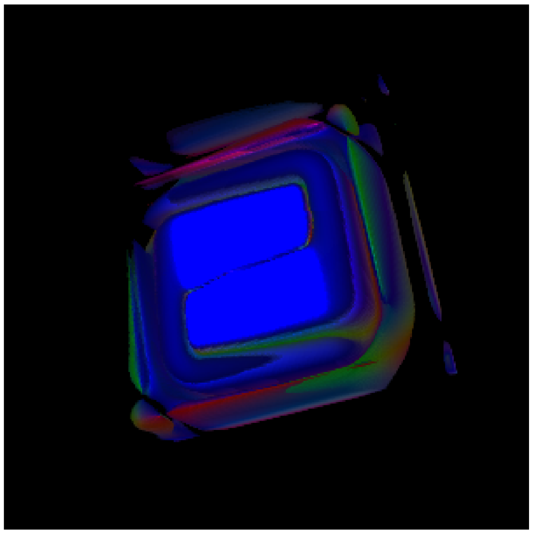
Fig. 10. Visualized execution results of 3D-FDTD electromagnetic simulation

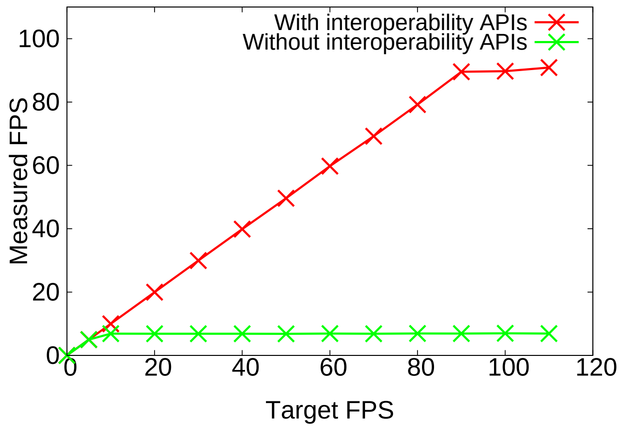

Fig. 11. Target FPS and measured FPS

usage rate is about 31 %. Therefore, also in terms of the device memory capacity, the triple-buffering technique was shown to be effective for practical simulation tasks.

VII. Conclusion

In this paper, we presented design and implementation of a fast runtime visualizer for a GPU-based 3D-FDTD electromagnetic simulation in order to increase the productivity of simulation development without compromising on simulation performance. The visualizer was designed with a software architecture pattern called the MVC model, so that general versatility was retained. By introducing the interoperability mechanism between OpenGL and CUDA, simulation results calculated by CUDA kernels are efficiently passed to programmable shaders, making high-throughput volume rendering possible. Multi-threading execution model was also introduced to offer a simple control flow to application developers and to prevent mutual performance interference between simulation kernels and visualization process.

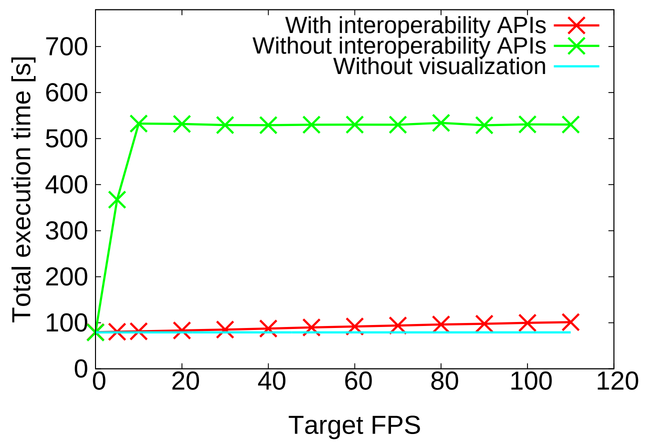

Fig. 12. Target FPS and execution time

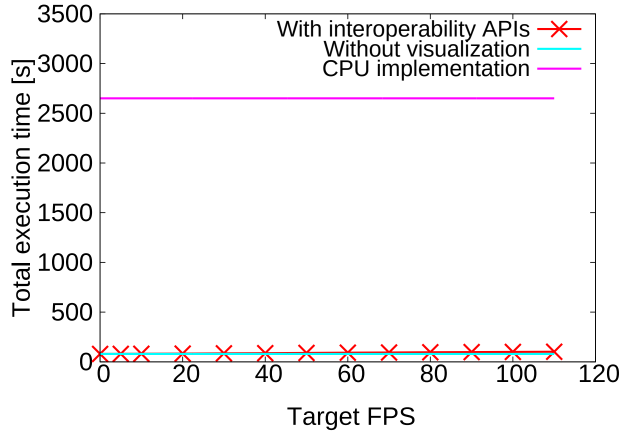

Fig. 13. Comparison of execution time

As a result of visualization experiments of a 3D-FDTD electromagnetic simulation for practical antenna characterization analysis, it was demonstrated that the maximum rendering throughput of 90 FPS was achieved for a viewport of 512×512 pixels. This result was a 12.9 times speedup compared to when the OpenGL-CUDA interoperability mechanism was not utilized. For the common visualization throughput of 60 FPS, the overhead of execution time imposed by the visualization process was 15.8 %. Since a 28.8 times speedup was obtained by GPU acceleration compared to a CPU-based implementation still in this case, this visualization overhead would be justified in terms of productivity.

Our challenging future work includes evaluation of the visualizer in terms productivity. Although it is difficult to evaluate the productivity in a quantitative way, some indirect assessments would be possible and worthy to try. Adding further visualization options and schemes is also important to increase the versatility of the visualizer.

References

[1] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. on antennas and propagation*, vol. 14, no. 3, pp. 302–307, 1966.

[2] A. Taflove and M. E. Brodwin, "Numerical solution of steady-state electromagnetic scattering problems using the time-dependent Maxwell's equations," *IEEE Trans. on Microwave Theory and Techniques*, vol. 23, no. 8, pp. 623–630, 1975.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12.

[4] D. Keisuke, Y. Shibata, K. Oguri, and T. Fujimoto, "Implementation of a gpu-oriented absorbing boundary condition for 3d-fdtd electromagnetic simulation," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 12, pp. 2787–2795, 2012.

[5] A. Sidelnik, S. Maleki, B. Chamberlain, M. Garzaran, and D. Padua, "Performance portability with the chapel language," in *Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2012, pp. 582–594.

[6] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for gpu program optimizations," in *Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2009, pp. 1–10.

[7] E. J. M. NORIEGA and T. NARUMI, "High performance n-body simulation and visualization through cuda architecture," *Bulletin of the University of Electro-Communications*, vol. 24, no. 1, pp. 59 – 64, 2012.

[8] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker, "A streaming narrow-band algorithm: interactive computation and visualization of level sets," vol. 10, no. 4, 2004, pp. 422–433.

[9] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical visualization and compression of large volume datasets using gpu clusters," in *Proc. of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2004, pp. 41–48.

[10] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting," in *Proc. of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics (VG)*, 2005, pp. 187–195.

[11] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A framework for computation and visualization on a gpu cluster," vol. 27, no. 2, pp. 341–350, 2008.

[12] E. Gobbetti, F. Marton, and J. A. Iglesias Guitian, "A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7, pp. 797–806, 2008.

[13] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 79–84.

[14] D. Shreiner, M. W. J. Neider, and T. Davis, "OpenGL programming guide: The official guide to learning OpenGL version2 (5th Edition)," 2006.

[15] Y. Kanamori, Z. Szego, and T. Nishita, "Gpu-based fast ray casting for a large number of metaballs," in *Computer Graphics Forum*, vol. 27, no. 2. Wiley Online Library, 2008, pp. 351–360.