# CUDA-OpenGL Interoperability to Visualize Electromagnetic Fields Calculated by FDTD

**Veysel Demir[1] and Atef Z. Elsherbeni[2]**

[1]Department of Electrical Engineering
Northern Illinois University, DeKalb, IL 60115, USA
vdemir@niu.edu

[2]Department of Electrical Engineering
The University of Mississippi, University, MS 38677, USA
atef@olemiss.edu

***Abstract*** ─ In this contribution, a compute unified device architecture (CUDA) implementation of a two-dimensional finite-difference time-domain (FDTD) program is presented along with the OpenGL interoperability to visualize electromagnetic fields as an animation while an FDTD simulation is running. CUDA, which runs on a graphics processing unit (GPU) card, is used for electromagnetic field data generation and image manipulation, while OpenGL is used to draw field distribution on the screen. Since CUDA and OpenGL both run on GPU and share data through common memory in the framebuffer, the CUDA-OpenGL interoperability is very efficient in visualization of electromagnetic fields. Step by step details of implementation of this interoperability are demonstrated.

***Index Terms*** ─ FDTD, GPGPU, visualization.

## I. INTRODUCTION

Recently, graphics processing units (GPUs) have become a viable alternative to multi-core central processing units (CPUs) for parallel processing architectures to perform high performance scientific computing. Due to the increasing demand from the scientific community, vendors have been improving both the hardware and the required software platforms, thus introducing a new generation of general purpose computing on graphic processing unit (GPGPU) cards.

Initially, the GPUs were not designed for general purpose computing and programming these cards required the use of programming platforms such as OpenGL, Brook [1], and High Level Shader Language (HLSL), which require a steep learning curve. Recently, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [2] development environment as a general purpose parallel computing architecture which makes GPU computing much easier. Developers can use C language to write functions that can achieve high performance on a CUDA enabled graphics processor.

The computational electromagnetics community as well has started to utilize the computational power of these cards, and in particular, several implementations of finite-difference time-domain (FDTD) method [3,4,5] have been reported by academic researchers and commercial software vendors including the implementations based on CUDA [6-11].

CUDA and OpenGL are two software platforms both of which operate on the GPU hardware, while their intended use are different; CUDA is suitable for improving the performance of data parallel computations, while OpenGL is for producing 2D and 3D computer graphics. While running a FDTD simulation, it is possible to capture electromagnetic fields and visualize them as an on-the-fly animation. If the FDTD calculations are performed on a graphics card, which is used also to perform OpenGL operations

*Submitted On: Oct. 26, 2011*
*Accepted On: Jan. 16, 2012*

1054-4887 © 2012 ACES

to display the fields, one can copy the field data from the graphics card memory (device memory) to the computer's main memory (host memory) that is processed by the CPU, process the data to create an image, and copy the image back to the GPU memory to display via OpenGL. It is possible to avoid the back and forth data transfer between the host and device memories, and perform all the processing required for the display on the graphics card by employing CUDA-OpenGL interoperability provided by CUDA. Performing field calculations and processing the fields to create images for visualization simultaneously can considerably slow down the FDTD simulations and hinder the efficiency. The goal is to balance the tradeoff between a fast simulation and a high quality and smooth visualization. In this context, as presented in this contribution, CUDA-OpenGL interoperability improves the simultaneous calculation and visualization efficiency significantly. An implementation of CUDA-OpenGL interoperability is presented in the subsequent sections.

## II. CUDA-OPENGL INTEROPERABILITY IN FDTD

### A. Integration of FDTD with GLUT

The FDTD method is an iterative method in which the progressions of electromagnetic fields in time are simulated in a time marching loop. The time marching loop typically consists of functions to update sources, update electric and magnetic fields, apply boundary conditions, and capture fields. Many times the captured fields can be displayed on the fly as an animation of the fields. Such a FDTD algorithm is illustrated in Fig. 1. It is usually straightforward to program this algorithm in a programming language where a programmer can simply call a built-in function to display the fields. An example is Matlab [12], in which several plotting functions, such as `plot` and `imagesc`, are provided to the programmer to display data while the program is running. However in many other languages, such as C++ and Fortran, such functions are not available and one has to program the details of visualization code as well. OpenGL has been one of the most popular platforms to facilitate programming with visualization.

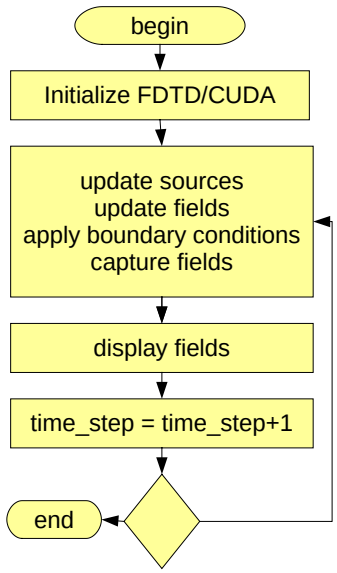


Fig. 1. FDTD algorithm.

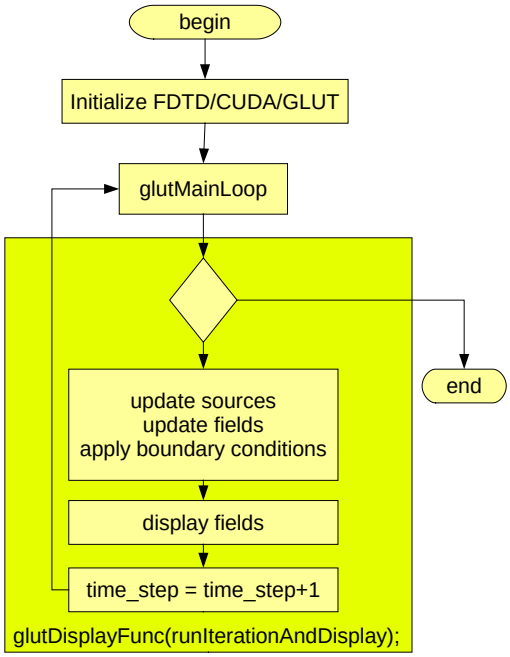


Fig. 2. FDTD algorithm integrated with GLUT.

Both CUDA and OpenGL can be programmed in C language. There are methods to integrate OpenGL in a program developed for an event driven operating system such as Microsoft Windows or Linux. One of the methods is to use GLUT. GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple

windowing application programming interface (API) for OpenGL [13]. In this contribution, CUDA/OpenGL interoperability is presented through the use of GLUT.

GLUT makes OpenGL programming simple yet platform independent, however, GLUT implements its own event loop. Therefore, mixing GLUT with an algorithm that demands its own event handling structure may be difficult. In order to integrate the FDTD algorithm with GLUT, the algorithm in Fig. 1 is modified as the one in Fig. 2. In this algorithm, first of all, FDTD, CUDA, and GLUT are initialized. CUDA-OpenGL interoperability also requires additional initialization at this stage as will be discussed. Then GLUT loop is started. Whenever GLUT loop triggers a display event, first a single iteration (or a number of iterations) of FDTD time marching loop is performed, and then results are displayed on a window.

**B. Initialization of OpenGL with CUDA**

CUDA is GPU programming platform developed and introduced by Nvidia. Nvidia provides extensive support to CUDA programmers. An article titled as "What Every CUDA Programmer Should Know about OpenGL" [14] is a good reference for beginners who want to learn CUDA-OpenGL interoperability. In this contribution, guidelines in [14] are followed to achieve interoperability between OpenGL and CUDA in an FDTD code. The details are presented in the steps below.

Listing 1. Initialization of FDTD, CUDA, and OpenGL

```
// global parameters
GLuint pbo_destination;
struct cudaGraphicsResource
*cuda_pbo_destination_resource;
GLuint cuda_result_texture;

bool runFdtdWithFieldDisplay
      (int argc, char** argv)
{
// Initialize CUDA context
cudaGLSetGLDevice
(cutGetMaxGflopsDeviceId());

// Initialize GL context
initializeGL(argc, argv);

// Initialize GL buffers
initializeGLBuffers();

// colormap used to map field
intensity
createColormapOnGpu();

// Display list of objects in problem
space
createDisplayListForObjects();

// copy data from CPU RAM to GPU
global memory
copyFdtdArraysToGpuMemory();

glutMainLoop();   // GLUT loop
}
```

Listing 2. Creating GL context

```
void initializeGL
      (int argc, char **argv )
{
setImageAndWindowSize();

// Create GL context
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA |
      GLUT_ALPHA | GLUT_DOUBLE |
      GLUT_DEPTH);
glutInitWindowSize(window_width,
      window_height);
iGLUTWindowHandle =
      glutCreateWindow
            ("CUDA OpenGL FDTD");

// initialize necessary OpenGL
extensions
glewInit();

// Initialize GLUT event functions
glutDisplayFunc
      (runIterationAndDisplay);
glutKeyboardFunc(keyboard);
glutReshapeFunc(reshape);
glutIdleFunc(idle);
}
```

*Initialize CUDA*

Listing 1 shows a function in which several functions are called to initialize FDTD, CUDA, and OpenGL and then FDTD simulations are started through GLUT. The first step is to initialize CUDA: the GPU device with maximum Gflops is set as the active device to run the FDTD calculations by a call to the function `cudaGLSetGLDevice()`.

*Initialize GL and Create a Window*

The next step is initialization of OpenGL and GLUT and creation of a window to display captured electromagnetic fields. GL initialization is performed in the function `initializeGL()` shown in Listing 2. The first part of Listing 2 initializes the GLUT. Next is an important step for CUDA-OpenGL interoperability in which OpenGL extensions are loaded to support buffers by calling the function `glewInit()` in Listing 2. Then event functions for GLUT are defined, thus GL initialization is completed. Here one should notice that `glutDisplayFunc()` is defined as `runIterationAndDisplay()`. As will be discussed later, when GLUT triggers a display event, the `runIterationAndDisplay()` will be executed, which will perform an iteration of FDTD time marching loop and display the electromagnetic field distribution in the problem space.

Listing 3. Initializing GL buffers

```
void initializeGLBuffers()
{
// create pixel buffer object
createPixelBufferObject
(&pbo_destination,
&cuda_pbo_destination_resource);

// create texture that will receive
the result of CUDA
createTextureDestination
      (&cuda_result_texture,
      image_width, image_height);
}
```

Listing 4. Creating pixel buffer object

```
void createPixelBufferObject(GLuint*
pbo, struct cudaGraphicsResource
**pbo_resource)
{
unsigned int texture_size =
sizeof(GLubyte) * image_width *
image_height * 4;

void *data = malloc(texture_size);

// create buffer object
glGenBuffers(1, pbo);
glBindBuffer(GL_ARRAY_BUFFER, *pbo);
glBufferData(GL_ARRAY_BUFFER,
texture_size, data, GL_DYNAMIC_DRAW);
free(data);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// register this buffer object with
CUDA
cudaGraphicsGLRegisterBuffer
(pbo_resource, *pbo,
cudaGraphicsMapFlagsNone);
}
```

Listing 5. Create texture

```
void createTextureDestination
(GLuint* cuda_result_texture,
unsigned int size_x,
unsigned int size_y)
{
// create a texture
glGenTextures(1,cuda_result_texture);
glBindTexture(GL_TEXTURE_2D,
*cuda_result_texture);

// set basic parameters
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexImage2D(GL_TEXTURE_2D, 0,
GL_RGBA8, size_x, size_y, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
}
```

*Create an OpenGL Buffer*

CUDA and OpenGL will use common resources on GPU for interoperability. Basically, these resources are buffers on the GPU's memory space. These buffers shall be created and initialized. In Listing 1, `initializeGLBuffers()` function, shown in Listing 3, is called for buffer initialization. Implementation of `initializeGLBuffers()` is shown in Listing 3. First, `createPixelBufferObject()` function, shown in Listing 4, is called, which creates a pixel buffer object and allocates memory for this buffer. The buffer will hold image data. The image is a field distribution in a two dimensional problem space composed of $n_{xx} \times n_{yy}$ cells. The field in each cell will be displayed with a single pixel, thus the image size is `image_width * image_height`, where `image_width =`

`nxx; image_height = nyy;`. Each pixel will hold red, green, blue, and alpha (RGBA) value of the pixel, thus each pixel uses four bytes of memory. Thus, the allocated memory is `4 * image_width * image_height`. CUDA will create the image and write to this buffer through the pixel buffer object, and then OpenGL will access to the same memory space and process it as a texture and display the image.

The texture as well needs to be initialized. The `createTextureDestination()`, shown in Listing 5, is used to initialize the texture.

### *Register Buffers for CUDA*

The last step in initialization of the pixel buffer object is to register the created buffer for CUDA. This is done in the last line of Listing 4 by calling the `cudaGraphicsGLRegisterBuffer()`. This command simply informs the OpenGL and CUDA drivers that this buffer will be used by both.

Listing 6. Display function of GLUT

```
void runIterationAndDisplay()
{
// run an FDTD iteration on GPU using
CUDA
for (int i=0; i< plotting_step; i++)
if (time_step<number_of_time_steps)
      fdtdIterationOnGpu();
      else
      {
      fetchResultsFromGpuMemory();
      deallocateArrays();
      saveSampledFieldsToFile();
      Cleanup(EXIT_SUCCESS);
      }

// Create image of field using CUDA
unsigned int* image_data;
// map the GL buffer to CUDA
cudaGraphicsMapResources(1,
&cuda_pbo_destination_resource, 0);
cudaGraphicsResourceGetMappedPointer
((void **)&image_data,
&number_of_bytes,
cuda_pbo_destination_resource);

// execute CUDA kernel
createImageOnGpu(image_data);
// unmap the GL buffer
cudaGraphicsUnmapResources(1,
&cuda_pbo_destination_resource, 0);

// Create a texture from the buffer
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_A
RB, pbo_destination);
glBindTexture(GL_TEXTURE_2D,
cuda_result_texture);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0,
0, image_width, image_height,
GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB
, 0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_A
RB, 0);

// draw the image
displayTextureImage
      (cuda_result_texture);

cudaThreadSynchronize();

// swap the front and back buffers
glutSwapBuffers();
}
```

Listing 7. Updating electric fields

```
__global__ void
update_electric_fields_on_kernel_TMz
(float* Ceze, float* Cezhy, float*
Cezhx, float* Hx,  float* Hy,
float* Ez, int nxx)
{
__shared__ float
sHy[TILE_SIZE][2*TILE_SIZE+1];

int tx = threadIdx.x;
int ty = threadIdx.y;
int i = blockIdx.x * blockDim.x + tx;
int j = blockIdx.y * blockDim.y + ty;

int ci = (j+1)*nxx+i;

sHy[ty][tx+TILE_SIZE] = Hy[ci];
sHy[ty][tx] = Hy[ci-TILE_SIZE];

__syncthreads();
Ez[ci] = Ceze[ci] * Ez[ci] +
Cezhy[ci] * (sHy[ty][tx+TILE_SIZE]-
sHy[ty][tx+TILE_SIZE-1]) + Cezhx[ci]
* (Hx[ci]-Hx[ci-nxx]);
}
```

Listing 8. Launch kernel to create the image

```
extern "C" void
createImageOnGpu
(unsigned int* image_data)
{
dim3 block(TILE_SIZE, TILE_SIZE, 1);
```

```
dim3 grid(nxx/block.x,
      nyy/block.y, 1);
createImageOnKernel
<<< grid, block>>>(image_data, dvEz,
nxx, min_value, max_value);
}
```

Listing 9. The kernel to create the image

```
__global__ void
createImageOnKernel(unsigned int*
image_data, float* Ez, int nxx,
float minval, float maxval)
{
int i = blockIdx.x * blockDim.x +
threadIdx.x;
int j = blockIdx.y * blockDim.y +
threadIdx.y;
int color_ind; float F;
int ci = j*nxx+i;
int ti = (j+1)*nxx+i;

F = Ez[ti] - minval;
color_ind = floor(255 * F/(maxval-
minval));
image_data[ci] = dvrgb[cind];
}
```

Listing 10. Displaying the image using OpenGL

```
void displayTextureImage
(GLuint texture)
{
glBindTexture(GL_TEXTURE_2D,
texture);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE, GL_REPLACE);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(domain_min_x,
domain_min_x+nxx*dx, domain_min_y,
domain_min_y+nyy*dy, -1.0, 1.0);

glMatrixMode( GL_MODELVIEW);
glViewport(0, 0, window_width,
window_height);

glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(domain_min_x,
      domain_min_y, 0.0);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(domain_min_x+nxx*dx,
      domain_min_y, 0.0);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(domain_min_x+nxx*dx,
      domain_min_y+nyy*dy, 0.0);
      glTexCoord2f(0.0, 1.0);
      glVertex3f(domain_min_x,
      domain_min_y+nyy*dy, 0.0);
glEnd();

glDisable(GL_TEXTURE_2D);

glCallList(objects_display_list);
}
```

### *Copy Colormap to GPU Memory*

After OpenGL and CUDA initializations are completed in Listing 1, a colormap is constructed on the GPU constant memory in `createColormapOnGpu()`. The colormap is basically a one dimensional array of 4 bytes integer including RGB values of colors that begin with blue, and pass through cyan, yellow, orange, and red. The colormap will be used to reflect the intensity of the displayed fields. The colormap array will be accessed at every iteration of the FDTD time marching loop in a CUDA kernel to generate the image of field distribution; thus, its access by CUDA has to be fast. To achieve fast access, the colormap array is stored in the constant memory of GPU.

### *Create a Display List of Objects*

It is possible to display an outline of objects that exist in the problem space as polygons together with the field distribution. An OpenGL display list is created in the function `createDisplayListForObjects()`. This display list is drawn on the field distribution image at every iteration, so it is more efficient to create it once at the beginning and use it during the time marching loop.

### *Copy FDTD Arrays to GPU Memory*

A CUDA program is a hybrid code which mainly runs on CPU, while parallel processing sections run on GPU. Therefore, the FDTD problem space, i.e. coefficient arrays, and field arrays, are initially constructed and allocated on the CPU RAM. These arrays need to be copied to GPU global memory to have them available for CUDA computations on GPU. These arrays are copied to GPU global memory in the

`copyFdtdArraysToGpuMemory()` function in Listing 1.

### C. Field calculations using CUDA and visualization of fields

As discussed before, whenever the display event (`glutDisplayFunc()`) is triggered in the GLUT loop, the associated function `runIterationAndDisplay()` is executed. Implementation of this function is shown in Listing 6. In Listing 6, a check is performed to see if the FDTD iterations are complete. If the iterations are complete, results are copied from the GPU global memory to the CPU memory, other post-processing operations are performed, and FDTD simulation is ended. If iterations are not complete, a number of iterations of the FDTD time marching loop are performed on GPU using CUDA by a call to the `fdtdIterationOnGpu()` function. An iteration includes usual steps such as an update of sources, an update of electric fields and magnetic fields, application of boundary conditions, and capture of electromagnetic fields. For instance, Listing 7 shows the kernel function that updates the electric field for the $TM_z$ case.

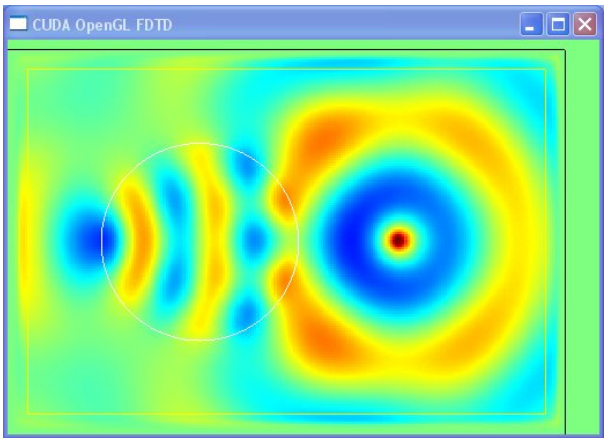


Fig. 3. A snapshot from the 2D FDTD program display.

After the field calculations, the new field distribution can be displayed on the created window, however, the field data cannot be displayed directly; data needs to be converted to an image first and the image needs to be stored in a texture in a form to be used by OpenGL. First, the GL buffer is mapped to CUDA, as shown in Listing 6, such that CUDA can process the field data and create image data. A GPU kernel function needs to be called to process the field data, thus the `createImageOnGpu()` function is called to launch the kernel function. Implementation of `createImageOnGpu()` is shown in Listing 8. The kernel function is `createImageOnKernel()` and is shown in Listing 9. This kernel basically maps the field value at each cell from a range between a minimum and a maximum to one of the 256 colors in the colormap, and stores to the image buffer.

Once the image is created in the buffer, the buffer is unmapped and released from CUDA. Then a texture is created from this buffer as shown in Listing 6. Once the texture is created, it is ready to get displayed by OpenGL. The function `displayTextureImage()`, shown in Listing 10, is called to perform the final display operations. Then `glutSwapBuffers()` is executed to show the image on the screen.

Figure 3 shows a snapshot from an animation of a two-dimensional FDTD simulation. The image is generated through the CUDA-OpenGL interoperability.

## III. PERFORMANCE OF CUDA-OPENGL INTEROPERABILITY

Two parametric sweep tests are performed by running the presented FDTD code in different modes to assess the performance improvement provided by CUDA-OpenGL interoperability. The following four modes are considered:

1. program is run on CPU only without field visualization
2. program is run on GPU using CUDA without field visualization
3. program is run on GPU using CUDA with field visualization and with CUDA-OpenGL interoperability
4. program is run on GPU using CUDA with field visualization, but without CUDA-OpenGL interoperability (the image data is transferred from device memory to host memory and displayed using OpenGL)

The analyses are performed on an NVIDIA® Tesla™ C1060 Computing Processor installed on a 64 bit Windows XP computer. This card has 240 streaming processor cores operating at 1.3 GHz.

The CPU results obtained using an Intel Xeon processor at 2 GHz.

Here, it should be noted that it is not necessary to display the images at every time step of the FDTD loop. Often it is sufficient to display a frame after a few time steps. A parameter denoted as `plotting_step`, shown in Listing 6, is used to control how often the images are displayed.

As the first parameter sweep test, the problem size of the two-dimensional space is increased, and each time the simulations are performed for the four modes with 1 frame per 5 iterations rate. Then, the throughputs of the simulations are calculated as [15]

$$NMCPS = \frac{n_{steps} \times n_{xx} \times n_{yy}}{t_s} \times 10^{-6}, \qquad (1)$$

where *NMCPS* is the number of million cells processed per second, $n_{steps}$ is the total number of time steps the program has been run, and $t_s$ is the total computation time in seconds. Here, $n_{xx}$ and $n_{yy}$ are the number of cells in an FDTD problem space in $x$ and $y$ directions, respectively. The results are shown in Fig. 4. Throughput is a measure of how fast the computations are performed, thus it can be used to assess the efficiency of the codes. The computation on GPU without any field visualization is much faster, as expected. Computation on CPU, even without visualization, is not comparable with GPU computation. In the presented Fig. 4, the computation with visualization is faster by 30% with CUDA-OpenGL interoperability. The results verify the efficiency improvement achieved by interoperability.

In the second parameter sweep test, the frame display rate is reduced (the image frames are displayed less often) and each time the simulations are performed for the four modes for a problem size of 6 million cells. The throughput results are shown in Fig. 5. Results again verify that visualizations are considerably faster with CUDA-OpenGL interoperability. The data transfers from the device memory to the host memory take considerable time when CUDA-OpenGL interoperability is not utilized. Moreover, as image display rate is reduced, the computations on GPU with visualizations converge to that of GPU without visualization.

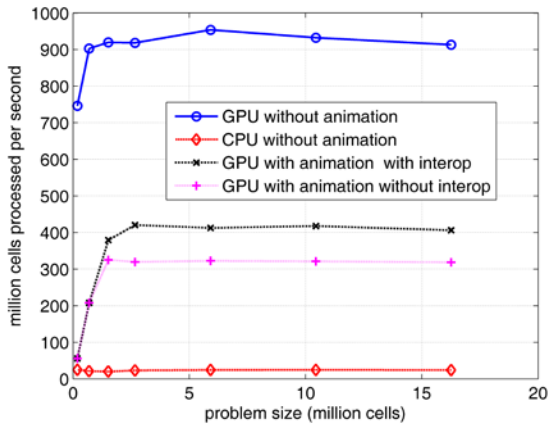


Fig. 4. Throughputs of different modes vs. problem size. 1 frame displayed per 5 time steps.

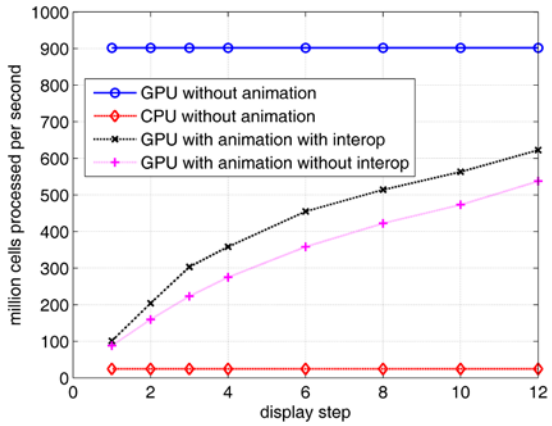


Fig. 5. Throughputs of different modes vs. image display rate. Problem size is 6 million cells.

The results in Fig. 5 also show that displaying the frames more often slows down the computations significantly. It is not necessary to display the frames very often. It has been observed that it is sufficient to display one frame per 5-10 iterations to achieve a smooth visual animation simultaneously with fast computation.

## IV. CONCLUSION

An implementation of CUDA-OpenGL interoperability to visualize electromagnetic fields in a two-dimensional FDTD simulation is developed and presented. It is shown that interoperability can improve the visualization efficiency significantly. Interoperability can be extended to three-dimensional FDTD and more complicated field visualizations can be achieved.

## REFERENCES


[1] I. Buck, *Brook Spec v0.2*, Stanford Univ. Press, 2003.

[2] NVIDIA CUDA ZONE: www.nvidia.com/object/cuda_home.html.

[3] K. S. Yee, "Numerical Solution of Inital Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, May 1966.

[4] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd edition, Artech House, 2005.

[5] A. Elsherbeni and V. Demir, *The Finite Difference Time Domain Method for Electromagnetics: with MATLAB Simulations*, *SciTech Publishing,* 2009.

[6] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to Render FDTD Computations More Effective Using a Graphics Accelerator," *IEEE Transactions on Magnetics*, vol. 45, no. 3, pp. 1324-1327, 2009.

[7] V. Demir and A. Z. Elsherbeni, "Compute Unified Device Architecture (CUDA) Based Finite-Difference Time-Domain (FDTD) implementation," *Journal of the Applied Computational Electromagnetics Society (ACES)*, vol. 25, no. 4, pp. 303-314, April 2010.

[8] C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. C. Hughes, C. Whelan, and M. Okoniewski, "Speed it Up," *IEEE Microwave Magazine*, vol. 11, no. 2, pp. 70-78, April 2010.

[9] M. Ujaldon, "Using GPUs for Accelerating Electromagnetic Simulations," *Journal of the Applied Computational Electromagnetics Society (ACES)*, vol. 25, no. 4, pp. 294-302, April 2010.

[10] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "Improved Performance of FDTD Computation Using a Thread Block Constructed as a Two-Dimensional Array with CUDA," *Journal of the Applied Computational Electromagnetics Society (ACES)*, vol. 25, no. 12, pp. 1061-1069, December 2010.

[11] M. R. Zunoubi and J. Payne, "Analysis of 3-Dimensional Electromagnetic Fields in Dispersive Media using CUDA," *Progress In Electromagnetics Research M*, vol. 16, pp. 185-196, 2011.

[12] http://www.mathworks.com.

[13] M. J. Kilgard, "The OpenGL Utility Toolkit (GLUT) Programming Interface, API Version 3". Silicon Graphics, Inc. November 13, 1996.

[14] J. Stam, "What Every CUDA Programmer Should Know About OpenGL," in GPU Technology Conference, San Jose, CA, October 1, 2009.

[15] Acceleware: www.acceleware.com


**Veysel Demir** is an Assistant Professor at The Department of Electrical Engineering, Northern Illinois University. He received his B.Sc. degree in Electrical Engineering from Middle East Technical University, Ankara, Turkey, in 1997. He studied at Syracuse University, New York, where he received both a M.Sc. and Ph.D. in Electrical Engineering in 2002 and 2004, respectively. During his graduate studies, he worked as research assistant for Sonnet Software, Inc., Liverpool, New York. He worked as a visiting research scholar in the Department of Electrical Engineering at the University of Mississippi from 2004 to 2007. He joined Northern Illinois University in August 2007. His research interests include numerical analysis techniques as well as microwave and radiofrequency (RF) circuit analysis and design. Dr. Demir is a member of IEEE and ACES and has coauthored more than 20 technical journal and conference papers. He is the coauthor of the books *Electromagnetic Scattering Using the Iterative Multiregion Technique* (Morgan & Claypool, 2007) and *The Finite Difference Time Domain Method for Electromagnetics with MATLAB Simulations* (SciTech 2009).

**Atef Z. Elsherbeni** is a Professor of Electrical Engineering and Associate Dean for Research and Graduate Programs, the Director of The School of Engineering CAD Lab, and the Associate Director of The Center for Applied Electromagnetic Systems Research (CAESR) at The University of Mississippi. In 2004, he was appointed as an adjunct Professor, at The Department of Electrical Engineering and Computer Science of the L.C. Smith College of Engineering and Computer Science at Syracuse University. On 2009, he was selected as Finland Distinguished Professor by the Academy of Finland and TEKES. Dr. Elsherbeni is a Fellow member of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow member of The Applied Computational Electromagnetics Society (ACES). He is the Editor-in-Chief for ACES Journal and an Associate Editor to the Radio Science Journal.