Progress In Electromagnetics Research, Vol. 116, 441–456, 2011

IMPLEMENTATION OF THE FDTD METHOD BASED ON LORENTZ-DRUDE DISPERSIVE MODEL ON GPU FOR PLASMONICS APPLICATIONS

K. H. Lee, I. Ahmed*, R. S. M. Goh, E. H. Khoo, E. P. Li, and T. G. G. Hung

Institute of High Performance Computing, 1 Fusionopolis Way, #16-16 Connexis, Singapore 138632, Singapore

Abstract—We present a three-dimensional finite difference time domain (FDTD) method on graphics processing unit (GPU) for plasmonics applications. For the simulation of plasmonics devices, the Lorentz-Drude (LD) dispersive model is incorporated into Maxwell equations, while the auxiliary differential equation (ADE) technique is applied to the LD model. Our numerical experiments based on typical domain sizes as well as plasmonics environment demonstrate that our implementation of the FDTD method on GPU offers significant speed up as compared to the traditional CPU implementations.

1. INTRODUCTION

A number of numerical methods have been developed for the simulation of electromagnetic applications in both frequency and time domain [1, 2]. Frequency domain methods such as finite element method (FEM) [3], method of moments (MoM) [4] are steady state techniques and are efficient for narrowband applications. Time domain methods such as FDTD [5], alternating direction implicit finite difference time domain (ADI-FDTD) [6], locally one dimensional finite difference time domain (LOD-FDTD) [7] are transient techniques and are efficient for wideband applications. In this paper, our focus is on the FDTD method. This method has attracted much attention due to its simplicity, accuracy, robustness, and its capability both in treating non-linear behavior naturally and providing real-time visualization response. In addition, this method has been applied to many areas such as electromagnetics, elastodynamics, photonics, RF/microwaves, biosensors, plasmonics and nanotechnology [8–13, 31].

Received 20 April 2011, Accepted 11 May 2011, Scheduled 16 May 2011

* Corresponding author: Iftikhar Ahmed (iahmed@ihpc.a-star.edu.sg).

However, with all these flexibilities and capabilities, the FDTD method requires very long simulation time for structures that require large number of fine meshes. In addition to various parallel processing techniques, two hardware accelerator approaches have been proposed recently to enhance the simulation speed: i) field programmable gate arrays (FPGAs) [14], ii) graphics processing unit (GPU) [15]. GPU, as compared to the FPGA, is garnering more traction due to its lower cost and its prevalent availability in mainstream computers. With easy access to this resource, it is relatively easy to test and implement different numerical techniques on the GPUs. The initial implementation of GPU for numerical computation was tedious and time-consuming, primarily because the initial design of GPU was only for graphics applications. In 2006 the CUDA technology was introduced by Nvidia, which lowered the learning curve to program and utilize the GPUs. This new concept supports FDTD type of algorithms which have natural characteristics of parallelization to run faster and accurately. Since then, GPU-accelerated FDTD method has been applied to different applications. In [16], the two-dimensional FDTD method is implemented on GPU for dispersive media using single pole Debye model with piecewise linear recursive convolution (PLRC) method for microwave applications. In [17], the three-dimensional FDTD method is implemented on GPU for low and mid frequency acoustics applications. In [18], two-dimensional FDTD using Drude model is implemented on GPU for double negative (DNG) materials. Optimization of the FDTD method for computation on heterogeneous and GPU clusters is presented in [19]. Similar to the FDTD method, GPU has also been used for the implementation of the finite difference frequency domain (FDFD) method for electromagnetic scattering applications [20].

In this paper, the FDTD method is implemented on GPU for plasmonics applications. According to our knowledge, this is the first paper on GPU for plasmonics applications using LD model. Plasmonics is an emerging area and deals with electromagnetic wave propagation at the interface of metal and dielectric. A number of plasmonics structures have been simulated and fabricated [21–23]. Because of dispersive nature of numerous metals at optical frequencies Lorentz-Drude (LD) dispersive model is incorporated into Maxwell's equations. The ADE approach is applied to LD model to make consistent with the FDTD method. As an example a nanosphere is studied. Numerical results obtained by using central processing unit (CPU) are compared with those obtained by GPU.

In addition to GPU, we developed the same algorithm on Matlab and C++, which we execute as an comparison on modern CPU. We then run representative numerical experiments for each version and compare the performance throughput across all the implementations. We also run additional tests to compare the accuracy of the CPU vs GPU implementations so as to ensure that accuracy is not compromised. These developed codes are based on the FDTD method and can be utilized for the development of plasmonics applications.

2. FORMULATIONS

At optical frequencies numerous metals show dispersive nature. In order to model them accurately, different dispersive models have been incorporated into Maxwell equations [13, 16–28]. Most of the available dispersive models are in frequency domain, so as to make them consistent with time domain methods different approaches such as recursive convolution (RC), piecewise linear recursive convolution (PLRC), z-transform and auxiliary differential equation (ADE) [13, 18, 24–26] are used. PLRC and ADE are most commonly used approaches due to their accuracy and efficiency. PLRC is an integral approach and numerical convolution is needed for its implementation, whereas the ADE is a differential approach. In this paper, we use the Lorentz-Drude dispersive model and to further simplify it, ADE approach is used. The Lorentz-Drude model is written as

$$\varepsilon_r(\omega) = \varepsilon_\infty + \frac{\omega_{pD}^2}{j^2\omega^2 + j\Gamma_D\omega} + \frac{\Delta\varepsilon_L\omega_{pL}^2}{j^2\omega^2 + j\omega\Gamma_L + \omega_L^2} \tag{1}$$

In Equation (1), the second term denotes Drude model, while the third term denotes Lorentz model. In this formulation, we use only single pole for Lorentz model and is enough for the required accuracy. However, more number of poles can be used at the cost of simulation time.

Maxwell's equations in frequency domain can be written as

$$\nabla \times H = j\omega\varepsilon_0\varepsilon_r(\omega)E \tag{2}$$

$$\nabla \times E = -j\omega\mu_0\mu_r B \tag{3}$$

After putting Lorentz-Drude model into Equation (2), it becomes

$$\nabla \times H = j\omega\varepsilon_0 \left(\varepsilon_\infty + \frac{\omega_{pD}^2}{j^2\omega^2 + j\Gamma_D\omega} + \frac{\Delta\varepsilon_L\omega_{pL}^2}{j^2\omega^2 + j\omega\Gamma_L + \omega_L^2} \right) E \tag{4}$$

The Drude part in Equation (4) is written as

$$j\omega\varepsilon_0 \frac{\omega_{pD}^2}{j^2\omega^2 + j\Gamma_D\omega} E = J_D \tag{5}$$

The Lorentz part in Equation (4) is written as

$$\frac{\Delta\varepsilon_L\omega_{pL}^2}{j^2\omega^2+j\omega\Gamma_L+\omega_L^2}E=P_L \tag{6}$$

After putting (5) and (6) into Equation (4) we get

$$\nabla\times H=\varepsilon_0\varepsilon_\infty\frac{\partial E}{\partial t}+J_D+\frac{\varepsilon_0\partial P_L}{\partial t} \tag{7}$$

Whereas ADE approach is applied on Equations (5) and (6) to make them compatible with time domain Maxwell's equations. As an example equations of the electric and magnetic fields for the FDTD method in x-direction are written as
$\underline{H_x}$

$$\begin{aligned}H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right) &= H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)\\ &+\frac{\Delta t}{\mu}\left[\begin{array}{c}\frac{E_y^n\left(i,j+\frac{1}{2},k+1\right)-E_y^n\left(i,j+\frac{1}{2},k\right)}{\Delta z}\\ -\frac{E_z^n\left(i,j+1,k+\frac{1}{2}\right)-E_z^n\left(i,j,k+\frac{1}{2}\right)}{\Delta y}\end{array}\right]\end{aligned} \tag{8}$$

$\underline{E_x}$

$$\begin{aligned}&E_x^{n+1}\left(i+\frac{1}{2},j,k\right)\\ =\ &\frac{1}{\Omega_x}E_x^n\left(i+\frac{1}{2},j,k\right)+\frac{\Delta t}{\Omega_x\varepsilon_0\varepsilon_\infty}\left[\begin{array}{c}\frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y}\\ -\frac{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right)-H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z}\end{array}\right]\\ &-\frac{\Delta t}{2\Omega_x\varepsilon_0\varepsilon_\infty}\left[\alpha_xJ_x^n\left(i+\frac{1}{2},j,k\right)+\beta_xE_x^n\left(i+\frac{1}{2},j,k\right)+J_x^n\left(i+\frac{1}{2},j,k\right)\right]\\ &-\frac{\varepsilon_0}{\Omega_x\varepsilon_0\varepsilon_\infty}\left[\varsigma_xE_x^n\left(i+\frac{1}{2},j,k\right)+\tau_xP_x^n\left(i+\frac{1}{2},j,k\right)\right.\\ &\left.-\rho_xP_x^{n-1}\left(i+\frac{1}{2},j,k\right)-P_x^n\left(i+\frac{1}{2},j,k\right)\right]\end{aligned} \tag{9}$$

Drude model

$$\begin{aligned}&J_x^{n+1}\left(i+\frac{1}{2},j,k\right)\\ =\ &\alpha_xJ_x^n\left(i+\frac{1}{2},j,k\right)+\beta_x\left[E_x^{n+1}\left(i+\frac{1}{2},j,k\right)+E_x^n\left(i+\frac{1}{2},j,k\right)\right]\end{aligned} \tag{10}$$

Lorentz model

$$\begin{aligned} P_x^{n+1}\left(i+\frac{1}{2},j,k\right) &= \varsigma_x\left(E_x^{n+1}\left(i+\frac{1}{2},j,k\right)+E_x^{n}\left(i+\frac{1}{2},j,k\right)\right) \\ &\quad +\tau_x P_x^{n}\left(i+\frac{1}{2},j,k\right)-\rho_x P_x^{n-1}\left(i+\frac{1}{2},j,k\right) \end{aligned} \tag{11}$$

where

$$\alpha_x = \frac{\left(1-\frac{\Delta t\Gamma_D}{2}\right)}{\left(1+\frac{\Delta t\Gamma_D}{2}\right)}, \quad \beta_x=\frac{\frac{\Delta t\omega_{pD}^2\varepsilon_0}{2}}{\left(1+\frac{\Delta t\Gamma_D}{2}\right)}, \quad \Omega_x=\left(\frac{\varepsilon_0\varsigma_x}{\varepsilon_0\varepsilon_\infty}+1+\frac{\Delta t\beta_x}{2\varepsilon_0\varepsilon_\infty}\right),$$

$$\varsigma_x = \frac{\frac{\Delta t^2\Delta\varepsilon_L\omega_{pL}^2}{2}}{\left(1+\Delta t\Gamma_L+\frac{\Delta t^2}{2}\omega_L^2\right)}, \quad \tau_x=\frac{\left(2+\Delta t\Gamma_L-\frac{\Delta t^2}{2}\omega_L^2\right)}{\left(1+\Delta t\Gamma_L+\frac{\Delta t^2}{2}\omega_L^2\right)},$$

$$\rho_x = \frac{1}{\left(1+\Delta t\Gamma_L+\frac{\Delta t^2}{2}\omega_L^2\right)}$$

3. GPU IMPLEMENTATION

The current generation of graphics processing unit (GPU) has hundreds of processing cores. These processing cores are grouped into multiprocessors. Each multiprocessor contains 8 to 32 processing cores. For example, Nvidia Tesla C2050 has 14 multiprocessors of 32 processing cores each. In other words, this GPU has 448 processing cores. In order for a GPU to be utilized efficiently, thousands of processing threads have to be executed. A huge number of threads are needed to mitigate the effect of threads being stalled due to memory access latency.

These threads are created and organized at two levels. For each multiprocessor, a block of threads, ranging from 1 to 1024, is created and executed. It will make sense to have more threads than the number of processing cores in the multiprocessor, so that each processing core has at least 1 thread to execute and another thread to switch to when the current thread is stalled. At the next level, a number of such blocks are created to execute on all the multiprocessors. Similarly, it will be advantageous to have more blocks than the number of multiprocessor.

The other factor that affects the utilization of the GPU is related to memory access. A graphics processing unit is packaged on a board with its own memory, known as device memory. For example, Nvidia Tesla C2050 has 3 gigabyte of device memory. The provision of device memory allows the GPU to do computations without the

CPU intervention. However, in order to maximize the memory access throughput, it is important to maximize coalescing. Coalescing is a mechanism to reduce the number of memory access transactions. For example, if a warp of 32 threads requested for a sequential set of memory locations, this request can be coalesced into one memory access transaction.

A small section, 64 kilobyte on Nvidia Tesla C2050, of the device memory can be classified as constant memory. The constraint for constant memory is that it only allows read access. However, the advantage is that there is a 8 kilobyte cache on Nvidia Tesla C2050 per multiprocessor to reduce the memory access latency.

On the GPU, there is 48 kilobyte of on-chip memory per multiprocessor on the Nvidia Tesla C2050. This memory is known as the shared memory. As mentioned earlier, there is a block of threads that is executed on each multiprocessor. However, each thread has its own memory space which is not accessible from another thread. Therefore, in order to solve this problem, the shared memory is used to allow the threads to access a common pool of memory on each multiprocessor. Since the shared memory is on the same chip as the GPU, it is as fast as the register.

Now we will look at the code segments that are used to update H_x and E_x fields as an example. At the same time, we will elaborate on how memory coalescing, constant memory and shared memory are used collectively to improve the overall performance of the GPU implementation. Figure 1, shows the flow chart of the method, which consists of three parts: pre-processing, GPU kernel execution and post-processing. Pre-processing, post-processing and the conditional check are executed on the CPU, while the kernels are on the GPU.

Each thread updates the H_x equation on a unique $(i,\, j+\frac{1}{2},\, k+\frac{1}{2})$ location. There is a loop in this thread to iterate through the range of k values without varying the i and j values.

According to Equation (8), in order to update $H_x\,(i,\, j+\frac{1}{2},\, k+\frac{1}{2})$, there is one variable that is reused in the loop, another variable that is shared between two neighboring threads and two constants that are invariant when updating H_x for all the i, j and k values. These two constants are stored in the constant memory and the constant memory cache is used to reduce the memory latency.

The first two dependencies are $E_y\,(i, j+\frac{1}{2},\, k)$ and $E_y\,(i, j+\frac{1}{2},\, k+1)$. Since each thread is iterating through the range of k values, $E_y\,(i,\, j+\frac{1}{2},\, k+1)$ is also used in the next iteration. Therefore, it is beneficial to save the value in the register to be used in the next iteration.

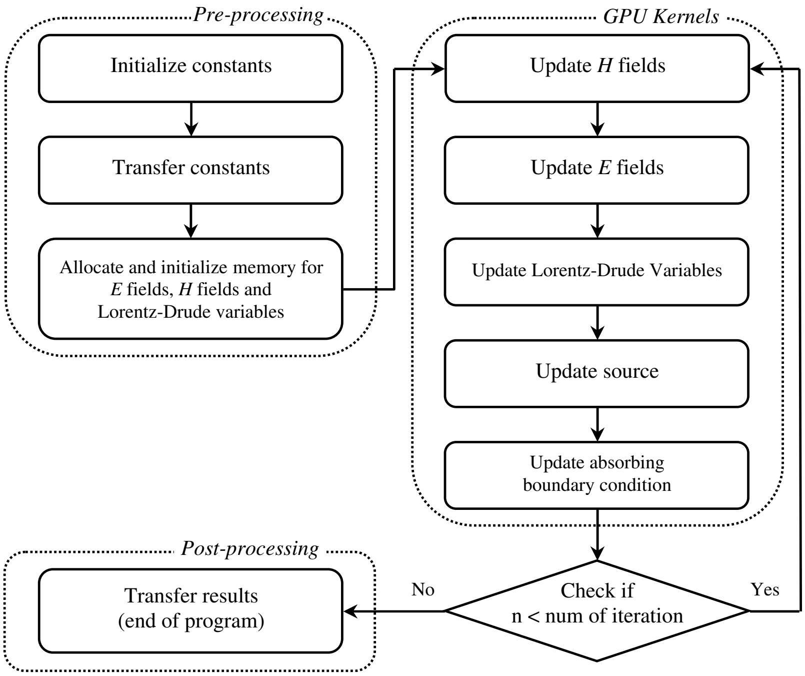
Pre-processing
Initialize constants
Transfer constants
Allocate and initialize memory for E fields, H fields and Lorentz-Drude variables
GPU Kernels
Update H fields
Update E fields
Update Lorentz-Drude Variables
Update source
Update absorbing boundary condition
Check if n < num of iteration
Yes
No
Post-processing
Transfer results (end of program)

Figure 1. Flow chart of GPU implementation.

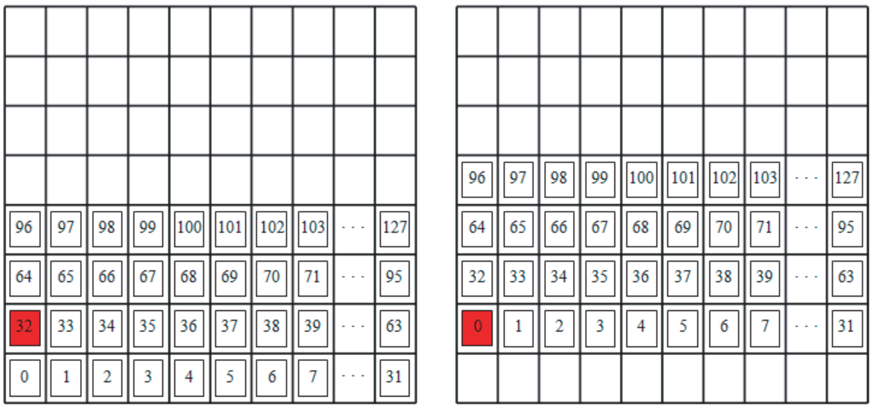

Figure 2. Left diagram shows the memory access pattern when accessing $E_z\left(i, j, k+\frac{1}{2}\right)$ and right diagram shows the memory access pattern when accessing $E_z(i, j+1, k+\frac{1}{2})$.

The other two dependencies are $E_z\,(i,\, j,\, k+\frac{1}{2})$ and $E_z\,(i,\, j+1,\, k+\frac{1}{2})$. Assuming that there is a block of 128 threads and accessing $E_z\,(i,\, j,\, k+\frac{1}{2})$ and $E_z\,(i,\, j+1,\, k+\frac{1}{2})$ as shown in Figure 2, it is clear that neighboring threads are sharing the memory access. For example, thread 0 and 32, as highlighted in the figure, are accessing the same E_z. Therefore, shared memory should be used to share the memory access between neighboring threads.

Also shown in Figure 2, the warp of 32 threads are accessing contiguous memory locations. Therefore, the memory access request is coalesced into one memory access transaction and this reduces the memory access latency. Figure 3 shows the code segment for updating the H_x field component.

```
__global__ void updateHx(double *hx, double *ey,
                                double *ez, int *material)
{
    double prevEy, currEy;
    extern __shared__ double s_Ez[];

    // Compute the increment and offset

    prevEy = ey[index];

    for (k=0; k<kEnd; k++)
    {
        currEy = ey[index+zIncr];

        s_Ez[sIndex] = ez[index];

        if (index < 32)
          s_Ez[sIndex+sIncr] = ez[index+yIncr];

        __syncthreads();

        hx[index] += c_chz[material[index]] *
                          (currEy – prevEy) –
                         c_chy[material[index]] *
                          (s_Ez[sIndex+offset] –
                           s_Ez[sIndex]);

        index += zIncr;
        prevEy = currEy;

        __syncthreads();
    }
}
```

Figure 3. Code segment for updating H_x field component.

On line 1, the "__global__" keyword is used to describe a function, also known as a kernel that is executed on the graphics processing unit.

On line 5, the "__shared__" keyword is used to declare a variable in shared memory. The "extern" keyword is used to indicate that the size is specified at run-time when this function is executed.

```
 1: __global__ void updateEx(double *oldEx, double *newEx,
 2:                         double *hy, double *hz,
 3:                         double *jx, double *oldLx,
 4:                         double *newLx, int *material)
 5: {
 6:   double prevHy, currHy;
 7:   extern __shared__ double s_Hz[];
 8:
 9:   // Compute the increment and offset
10:
11:    prevHy = hy[index-zIncr];
12:
13:    for (k=1; k<kEnd; k++)
14:    {
15:      currHy = hy[index];
16:
17:      s_Hz[sIndex] = hz[index];
18:
19:      if (index < 32)
20:        s_Hz[sIndex-sIncr] = hz[index-yIncr];
21:
22:      __syncthreads();
23:
24:      newEx[index] = c_ca[material[index]] * oldEx[index] +
25:                     c_cay[material[index]] *
26:                      (s_Hz[sIndex] - s_Hz[sIndex-offset]) -
27:                     c_caz[material[index]] *
28:                      (currHy - prevHy) -
29:                     c_cad[material[index]] * jx[index] -
30:                     c_caL1[material[index]] * oldLx[index] +
31:                     c_caL2[material[index]] * newLx[index];
32:
33:      jx[index] = c_alfa * jx[index] +
34:                  c_beta * (oldEx[index] + newEx[index]);
35:
36:      newLx[index] = c_tau * oldLx[index] –
37:                     c_rho * newLx[index] +
38:                     c_eta * (oldEx[index] + newEx[index]);
39:
40:      index += zIncr;
41:      prevHy = currHy;
42:
43:      __syncthreads();
44:    }
45: }
```

Figure 4. Code segment for updating the E_x field and Lorentz-Drude variables.

On line 9, it loads the current $E_y\,(i,\, j+\frac{1}{2},\, k)$ and it loads the next $E_y\,(i,\, j+\frac{1}{2},\, k+1)$ on line 13. It then stores the $E_y\,(i,\, j+\frac{1}{2},\, k+1)$ for the next iteration on line 29.

Line 11 describes a loop that iterates through the range of k values.

On line 15, each thread loads its E_z value into the shared memory. Subsequent lines 17 and 18 checks if the thread is the first block of 32 threads, then it loads the last block of 32 E_z values into the shared memory.

On line 20, it is the __syncthreads() function. This function acts as a barrier, waiting for all the threads in the block to reach this point before proceeding. This is necessary because the threads are using the E_z values in the shared memory. Therefore, it has to make sure that all the threads have loaded the E_z values into the shared memory before proceeding.

Lines from 22 to 26 describe the computation to update the value of H_x. In this computation, the variable c_chy and c_chz are needed. Since these variables are read-only, they are stored in the constant memory. With the use of constant memory cache, this helps to reduce the memory access latency.

Finally, there is another __syncthreads() function on line 31. This is to ensure that all the threads in the block have finished their computation before proceeding. This is necessary because new values of E_z will be loaded into the shared memory in the next iteration. Figure 4 shows the code segment for updating E_x field component and the Lorentz-Drude variables.

4. NUMERICAL RESULTS AND DISCUSSION

This section consists of two sub-sections, numerical results and performance of various programming models and platforms for the FDTD method. Section 4.1 is about the plasmonics application, and accuracy of the method with and without GPU. While the Section 4.2 is about the performance efficiency of the GPU as compared to the other programming models and platforms for the approach.

4.1. Numerical Results

For numerical validation and accuracy of the method with and without GPU for plasmonics applications, gold nanospheres of different radius are considered. Although different media around the sphere can be used, for simplicity in our application the surrounding media is air and a gold nanosphere with radius R is shown in Figure 5. To truncate the free space around the nanosphere, Mur absorbing

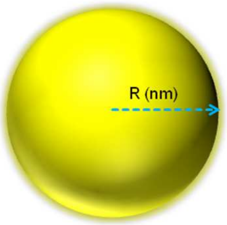

Figure 5. The illustration of gold nanosphere structure.

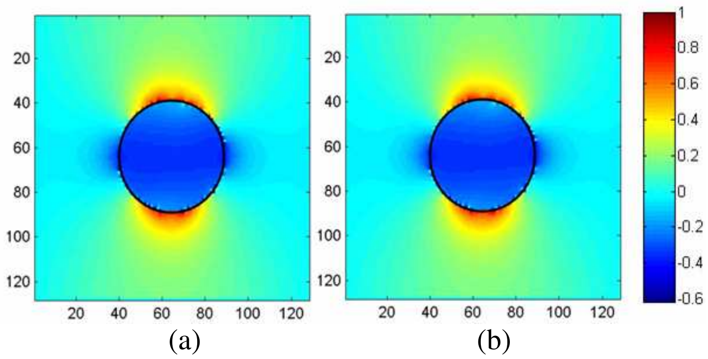

Figure 6. Electric field distribution inside and outside 25 nm gold nanosphere in free space (a) with GPU (b) without GPU.

boundary condition [29] is used. The same cell size is considered in all the three directions, i.e., $\Delta x = \Delta y = \Delta z = 1\,\text{nm}$. The parameters used for LD model are same as given in [30], i.e., $\varepsilon_\infty = 5.9673$, $\omega_{PD}/2\pi = 2113.6\,\text{THz}$, $\Gamma_D/2\pi = 15.92\,\text{THz}$, $\omega_{PL}/2\pi = 650.07\,\text{THz}$, $\Gamma_L/2\pi = 104.86\,\text{THz}$ and $\Delta\varepsilon_L = 1.09$.

The simulation was run until it reaches steady state. The electric field distribution both inside and outside of 25 nm gold nanosphere in free space is depicted in Figure 6. In Figure 6(a) the field is obtained with GPU, while in Figure 6(b) is obtained with CPU. Both figs. illustrate similar electric field distribution. Figure 7 is plotted for electric field intensity with respect to wavelength for different radius of nanospheres with and without GPU. With the changes in radius, there is change in the resonance wavelength, but the simulation results with and without GPU are in very good agreement. However, a significant improvement in simulation efficiency is observed in GPU as shown in Figs. 8 and 9, Tables 1 and 2. For example with domain size of $128 \times 128 \times 128$ and 60000 number of simulation iterations, GPU took 10.87 minutes, while Matlab took 49.64 hours.

Table 1. Presents the time taken (seconds) to complete the number of simulation steps using different programming platform/language. The domain size is 64 × 64 × 64.

Iterations	200	1000	5000	10000	60000
Matlab	74.94	373.49	1858.05	3729.29	22910.77
GCC-O3	5.31	26.27	132.09	262.5	1576.85
ICC-O3	5.88	28.94	143.65	286.9	1722.99
GPU	0.68	1.87	7.88	15.39	90.47

Table 2. Presents the time taken (seconds) to complete the number of simulation steps using different programming platform/language. The domain size is 128 × 128 × 128.

Iterations	200	1000	5000	10000	60000
Matlab	604.72	2950.38	14688.61	29515.71	178716.48
GCC-O3	42.74	208.22	1056.27	2068.84	12412.52
ICC-O3	47.02	230.79	1151.97	2296.57	13774
GPU	3.24	11.93	55.36	109.66	652.2

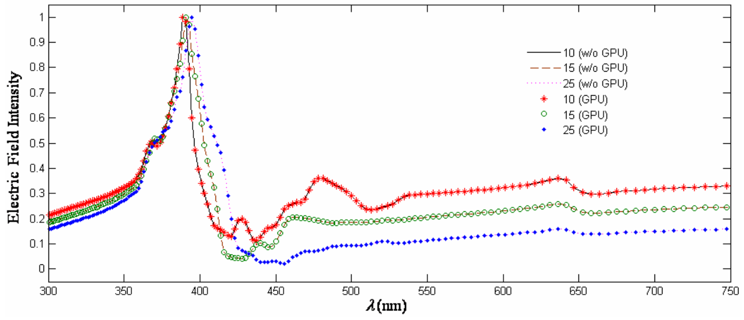

Figure 7. Electric field intensity with respect to wavelength for different sizes of the nanosphere with and without GPU.

4.2. Various Programming Models and Platforms

For further analysis and benchmarks, we developed the FDTD code by using different programming models and platforms, and compared their performance with GPU. A significant improvement in performance is observed with GPU, without affecting the accuracy of the results. The GPU card used is an Nvidia Tesla C2050, and the computer hardware specifications used in the numerical experiments is an Intel Core 2 Quad 3.2 GHz workstation with 4 GB RAM. The software and compilers used are: Matlab version R2008a, GCC 4.4.3 and Intel

Parallel Studio XE (v12.0). Two different domain sizes are tested, i.e., $64 \times 64 \times 64$ and $128 \times 128 \times 128$.

Figure 8 shows the performance bar graph of GPU, GCC-O3 and ICC-O3 with respect to number of simulation iterations, while the domain size is $64 \times 64 \times 64$. For this performance comparison, Matlab code is used as a reference. Figure 8 depicts that the GPU implementation outperforms the Matlab version by as much as 253 times. The GPU outperforms the serial C++ versions by about 19 times.

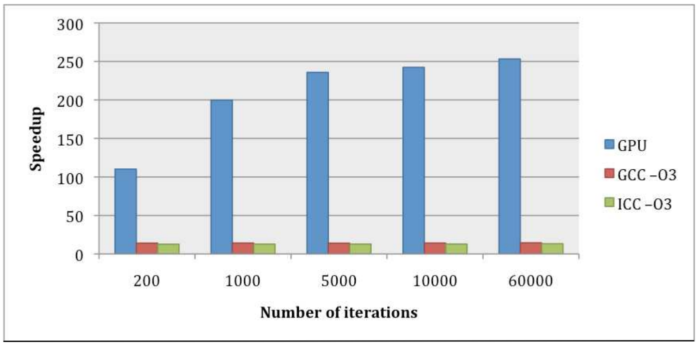

Figure 8. To reveal the speedup (number of times) of the GPU and C++ versions using different compilers as compared to the original Matlab version. The GPU version running on Nvidia Tesla C2050 outperforms Matlab version by as much as 253 times. The domain size is $64 \times 64 \times 64$.

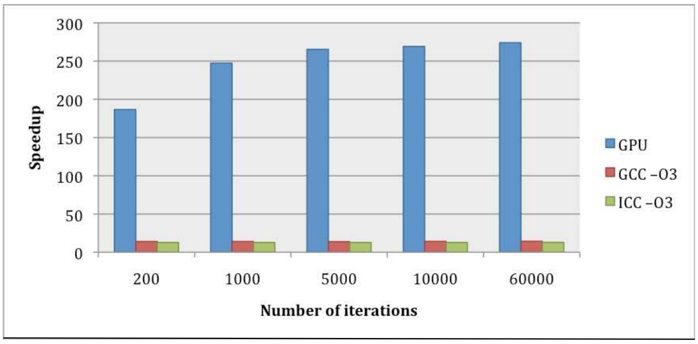

Figure 9. To reveal the speedup (number of times) of the GPU and C++ versions using different compilers as compared to the original Matlab version. The GPU version running on Nvidia Tesla C2050 outperforms Matlab version by as much as 274 times. The domain size is $128 \times 128 \times 128$.

Figure 9 shows the performance bar graph of GPU, GCC-O3 and ICC-O3 with respect to number of simulation time steps for larger domain size, i.e., $128\times128\times128$. The Matlab code is used as a reference for this comparison. As shown in Figure 9, the GPU implementation outperforms the Matlab version by as much as 274 times. The GPU outperforms the serial C++ versions by about 21 times.

5. CONCLUSION

We implemented a three-dimensional FDTD method on GPU for plasmonics applications. The method is based on Lorentz-Drude dispersive model which is incorporated into Maxwell equations, while the auxiliary differential equation technique is applied to the LD model. Through extensive correctness tests that are carried out, it has been shown evidently that our GPU implemented algorithm provides accurate results and are similar to the conventional version without GPU. The performance speed up that our CUDA-based FDTD method offers is up to 274 times faster than the Matlab version on CPU.

REFERENCES

1. Swanson, D. G. and W. J. R. Hofer, *Microwave Circuit Modeling Using Electromagnetic Field Simulation*, Artech House Inc., Norwood, MA, 2003.
2. Ahmed, I., E. H. Khoo, E. P. Li, and R. Mittra, "A hybrid approach for solving coupled Maxwell and Schrödinger equations arising in the simulation of nano-devices," *IEEE Antennas and Wireless Component Letters*, Vol. 9, 914–916, 2010.
3. Volakis, J. L., A. Chatterjee, and L. C. Kempel, *Finite Element Method for Electromagnetics: Antennas, Microwave Circuits, and Scattering Applications*, Wiley-IEEE, 1998.
4. Liu, Z. H., E. K. Chua, and K. Y. See, "Accurate and efficient evaluation of method of moments matrix based on a generalized analytical approach," *Progress In Electromagnetics Research*, Vol. 94, 367–382, 2009.
5. Yee, K. S., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propag.*, Vol. 14, 302–307, May 1966.
6. Zheng, F., Z. Chen, and J. Zhang, "A finite-difference time-domain method without the Courant stability conditions," *IEEE Microw. Guided Wave Lett.*, Vol. 9, No. 11, 441–443, 1999.
7. Ahmed, I., E. K. Chua, E. P. Li, and Z. Chen, "Development of the three dimensional unconditionally stable LOD-FDTD method," *IEEE Trans. Antennas Propag.*, Vol. 56, No. 11, 3596–3600, 2008.

8. Gaidamauskaite, E. and R. Baronas, "A comparison of finite difference schemes for computational modelling of biosensors," *Nonlinear Analysis: Modelling and Control*, Vol. 12, 359–369, 2007.
9. Yang, S., Y. Chen, and Z.-P. Nie, "Simulation of time modulated linear antenna arrays using the FDTD method," *Progress In Electromagnetics Research*, Vol. 98, 175–190, 2009.
10. Chen, J., "Application of the nearly perfectly matched layer for seismic wave propagation in 2D homogeneous isotropic media," *Geophysical Prospecting*, 2011, doi: 10.1111/j.1365-2478.2011.00949.x.
11. Ingo, W., "Finite difference time-domain simulation of electromagnetic fields and microwave circuits," *International Journal of Numerical Modelling*, Vol. 5, 163–182, 1992.
12. Seo, M., G. H. Song, et al., "Nonlinear dispersive three-dimensional finite-difference time-domain analysis for photonic-crystal lasers," *Opt. Express*, Vol. 13, 9645–9651, 2005.
13. Taflove, A. and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd edition, Artech House, Norwood, MA, 2005.
14. Maxfield, C., *The Design Warrior's Guide to FPGAs*, Elsevier, 2004.
15. Elsherbeni, A. Z. and V. Demir, *The Finite-Difference Time-Domain Method for Electromagnetics with Matlab Simulations*, SciTech Pub., 2009.
16. Zunoubi, M. R., J. Payne, and W. P. Roach, "CUDA implementation of TEz-FDTD solution of Maxwell's equations in dispersive media," *IEEE Antennas and Wireless Propagation Letters*, Vol. 9, 756–759, 2010.
17. Savioja, L., "Real-time 3D finite-difference time-domain simulation of low and mid frequency room acoustics," *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*, Graz, Austria, September 6–10, 2010.
18. Chen, S., S. Dong, and X.-L. Wang, "GPU-based accelerated FDTD simulations for double negative (DNG) materials applications," *International conference on Microwave and Millimeter Wave Technology (ICMMT)*, 839–841, 2010.
19. Shams, R. and P. Sadeghi, "On optimization of finite-difference time-domain (FDTD) computation on heterogeneous and GPU clusters," *J. Parallel Distrib. Comput.*, 2010.
20. Zainud-Deen, S. H., E. Hassan, M. S. Ibrahim, K. H. Awadalla, and A. Z. Botros, "Electromagnetic scattering using GPU

based finite difference frequency domain method," *Progress In Electromagnetics Research B*, Vol. 16, 351–369, 2009.
21. Maier, S. A., *Plasmonics: Fundamentals and Applications*, Springer-Verlag, 2007.
22. Ahmed, I., C. E. Png, E. P. Li, and R. Vahldieck, "Electromagnetic propagation in a novel Ag nanoparticle based plasmonic structure," *Opt. Express*, Vol. 17, 337–345, 2009.
23. Shalaev, V. M. and S. Kawata, *Nanophotonics with Surface Plasmons (Advances in Nano-Optics and Nano-Photonics)*, Elsevier, 2007.
24. Okoniewski, M., M. Mrozowski, and M. A. Stuchly, "Simple treatment of multi-term in FDTD," *IEEE Micro. Guided Wave Lett.*, Vol. 7, No. 5, May 1997.
25. Baumann, D., C. Fumeaux, C. Hafner, and E. P. Li, "A modular implementation of dispersive materials for time-domain simulations with application to gold nanospheres at optical frequencies," *Opt. Express*, Vol. 17, No. 17, 15186–15200, August 2009.
26. Shibayama, J., A. R. Nomura Ando, J. Yamauchi, and H. Nakano, "A frequency-dependent LOD-FDTD method and its application to the analyses of plasmonic waveguide devices," *IEEE Journal of Quantum Electronics*, Vol. 46, No. 1, 40–49, 2010.
27. Zhang, Y. Q. and D. B. Ge, "A unified FDTD approach for electromagnetic analysis of dispersive objects," *Progress In Electromagnetics Research*, Vol. 96, 155–172, 2009.
28. Wei, B., S.-Q. Zhang, Y.-H Dong, and F. Wang, "A general FDTD algorithm handling thin dispersive layer," *Progress In Electromagnetics Research B*, Vol. 18, 243–257, 2009.
29. Mur, G., "Absorbing boundary conditions for the finite difference approximation of the time domain electromagnetic field equations," *IEEE Transaction on Electromagnetic Compatibility*, Vol. 23, No. 4, 337–382, November 1981.
30. Vial, A., A. S. Grimault, D. Macias, D. Barchiesi, and M. D. Chapelle, "Improved analytical fit of gold dispersion Application to the modeling of extinction spectra with a finite-difference time-domain method," *Physical Review B*, Vol. 71, No. 8, 085416, February 2005.
31. Xu, K., Z. Fan, D.-Z. Ding, and R.-S. Chen, "Gpu accelerated unconditionally stable crank-nicolson FDTD method for the analysis of three-dimensional microwave circuits," *Progress In Electromagnetics Research*, Vol. 102, 381–395, 2010.