Accelerating the simulation of Finite Difference Time Domain (FDTD) with GPU

Yongjun Liu, Jing Wang, Jian Liu

Cadence Design Systems, Inc

San Jose CA

Abstract: In this paper, we discuss the implementation of three dimensional FDTD method to Nvidia's CUDA architecture. Finite-difference time-domain is a numerical analysis technique used for modeling computational electrodynamics. Since it is a time-domain method, FDTD solutions can cover a wide frequency range with a single simulation run, and treat nonlinear material properties in a natural way. Because FDTD requires the grid must be sufficiently fine to resolve both the smallest electromagnetic wavelength and the smallest geometrical feature in the model, it may result in very long simulation time. OpenMP, MPI, SIMD technique can be used to speed up the simulation. Another technology, Nvidia's CUDA, allows the time to be reduced a lot.

Keywords: Finite Difference Time Domain, FDTD, GPU, CUDA, CUDA-aware, Parallel

2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium | 978-1-6654-4888-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/EMC/SI/PI/EMCEurope52599.2021.9559260

1. Introduction

The FDTD method is well suited for analyzing problems with complex geometrical features as well as those containing arbitrarily inhomogeneous materials. Though the traditional FDTD technique does place a relatively heavy burden on computer resources when it is used to analyze a complex problem that occupies a large computational volume or includes very fine features. However, this problem can be solved by the parallel processing technique, SIMD, OpenMP and Message Passing Interface (MPI), that FDTD method is best fit to. Parallel FDTD method has become one of the major EM tools for large EM problems due to the fast development of computer science and low price of computer hardware. Nvidia's GPU architecture, Compute Unified Device Architecture (CUDA), provides a new way for parallel simulation.

In this paper, we discuss the challenges and techniques used to implement the FDTD algorithm in Nvidia's CUDA framework. Modern GPU evolved into highly parallel-core systems allowing very efficient manipulation of large block of data. This design is more effective than general purpose CPU to implement some algorithm where processing large block of data in parallel.

2. Finite Difference Time Domain

The FDTD method utilizes the central difference approximation to discretize the two Maxwell's curl equations, namely, Faraday's and Ampere's laws, in both the time and spatial domains, and then solves the resulting equations numerically to derive the electric and magnetic field distributions at each time step using an explicit leapfrog scheme. In Yee's scheme [1], the computational domain is discretized by using a rectangular grid. The electric fields are located along the edges of the electric elements, while the magnetic fields are sampled at the centers of the electric element surfaces and are oriented normal to these surfaces, this being consistent with the duality property of the electric and magnetic fields in Maxwell's equations. A typical electric unit is shown in Figure 1.

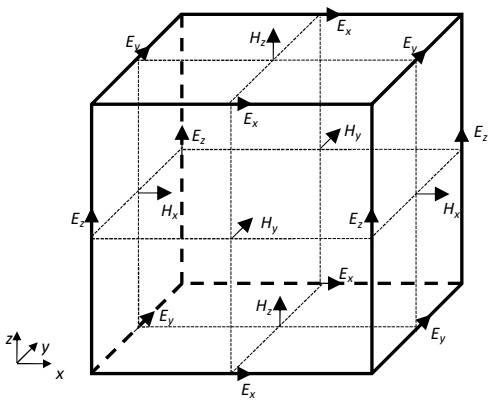

Figure 1: Yee Cell

The FDTD algorithm constructs a solution to the following two Maxwell's curl equations:

$$\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} - \sigma_M \vec{H} \qquad \text{(Faraday's law)}$$

$$\nabla \times \vec{H} = \varepsilon \frac{\partial \vec{E}}{\partial t} + \sigma \vec{E} \qquad \text{(Ampere's law)}$$

We can represent Maxwell's in the following explicit formats. The other components are similar.

$$E_x^{n+1}\left(i+\frac{1}{2},j,k\right) = \frac{\varepsilon_x - 0.5\Delta t\sigma_x}{\varepsilon_x + 0.5\Delta t\sigma_x} E_x^{n}\left(i+\frac{1}{2},j,k\right)$$

$$+ \frac{\Delta t}{\varepsilon_x + 0.5\Delta t\sigma_x}\left[\frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) - H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right) - H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z}\right]$$

3. Graphical Processing unit (GPU) and CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

CUDA C/C++ provides an abstraction; it's a means for you to express how you want your program to execute. The compiler generates PTX code which is also not hardware specific.

In GPU programming, there are two very important terminology: host and device.

Host: CPU and its memory (host memory)

Device: GPU and its memory (device memory)

Code running on host is called host code, code running on GPU is called device code. Simple CUDA Processing Flow works like below

1. Start host code
2. Copy input data from CPU memory to GPU memory
3. Load GPU program (device code) and execute, caching data on chip for performance
4. Copy results from GPU memory to CPU memory

```
__global__ void mykernel(void)
{
    printf("Hello World!\n");
}

int main(void)
{
    mykernel<<<1,1>>>();

    return 0;
}
```

Figure 2: Simple CUDA program

Here, mykernel() is a device function, it is called by host function main.

Applications can distribute work across multiple GPUs. This is not done automatically, however, the application has complete control. MPI, the Message Passing Interface, is a standard API for communicating data via messages between distributed processes that is commonly used in HPC to build applications that can scale to multi-node computer clusters. CUDA-aware MPI provides the ability to send and receive GPU buffers directly.

4. Implementation

GPU computing is about massive parallelism. We will implement those GPU-targeted function as kernels in CUDA, field update, boundary condition update, far field calculation, etc. Writing CUDA kernel function is very similar to writing C language. This make it very easy to transfer from C to CUDA. Considering the Ex field update in figure 2, implementing this in CUDA kernel is as simple as eliminating a triple for loops and placing the core function into one kernel call in figure 3.

```
for(i = 0; i <= Nx - 1; i++)
{
    for(j = 0; j <= Ny; j++)
    {
        for(k = 0; k <= Nz; k++)
        {
            dh1 = pEj_Coeff[j] * ( Hz[i][j][k] - Hz[i][j-1][k] );
            dh2 = pEk_Coeff[k] * ( Hy[i][j][k] - Hy[i][j][k-1] );

            Ex[i][j][k] =  eMatType[i][j][k]->AEx_Coeff * Ex[i][j][k]
                + eMatType[i][j][k]->AEx_p_Inv * ( dh1 - dh2 );
        }
    }
}
```

Figure 3: E field update implementation in C

We implement GPU kernel code for Ex field update as below.

```
__global__ void kernelUpdateEx(float *Ex, float *Hy, float *Hz,
    const float *pEj_Coeff, const float *pEk_CoeffDev,
    int nx, int ny, int nz, E_Material **eMatType)
{
    int index = computeIndex();
    int i = 0, j = 0, k = 0;

    int nyz = ny * nz;
    i = index / nyz;
    j = (index % nyz) / nz;
    k = (index % nyz ) % nz;


    int indexj = index - nz;
    int indexk = index - 1;

    if ( i <= nx - 1) && j <= ny && k <= nz)
    {
        dh1 = pEj_Coeff[j] * ( Hz[index] - Hz[indexj] );
        dh2 = pEk_Coeff[k] * ( Hy[index] - Hy[indexk] );

        Ex[index] = eMatType[index]->AEx_Coeff * Ex[index]
            + eMatType[index]->AEx_p_Inv * ( dh1 - dh2 );
    }
}
```

Figure 4: E field update implementation in CUDA

The final implementation includes E & H field updates, boundary condition (PEC, PMC and PML) updates, far field calculation, planewave source, dispersive material, SAR calculation, etc.

FDTD method is inherently parallel in nature, because we only need to pass the field on the interface between the adjacent subdomains. MPI is used for parallel implementation in our product. CUDA aware MPI is used for implementation for multiple GPUs. With CUDA-aware MPI, we only need to send GPU buffer instead of host buffer. With regular MPI, we use the following code to send and receive data as in figure 5.

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

Figure 5: MPI send and receive without CUDA aware

With CUDA-aware MPI, we use the following code to send and receive data as in figure 6.

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

Figure 6: MPI send and receive with CUDA aware

5. Performance test

We ran different test cases and hardware platform during implementation. We use a powerful GPU server for the benchmark. Our CPU implementation had utilized Intel SIMD technology, which is about 4 times faster than regular CPU programming. We compare the performance between GPU and Streaming SIMD Extensions (SSE, a single instruction, multiple data (SIMD) instruction set extension). Table 1 list the hardware and OS specification.

Table 1: System Specification

	GPU server
OS	Red Hat 7.6
GPU	Nvidia Tesla V100-SXM2-32GB x 4, each 5120 cores
CPU	Intel Xeon 6148 CPU @ 2.40GHz x 2, each 20 cores
CPU memory	1TB
GPU memory	32GB x 4 = 128GB

The timing comparisons are based on the user experience-meaning they include all the initialization, field update, boundary condition update, convergence check, data I/O. The test used all the 40 cores for CPU baseline and all the 5120 x 4 cores for the GPU, 4 cores for CPU implementation.

We compared two real case for benchmark test. Table 2 list the benchmark of a modern cell phone design. The simulation included all major components, antenna, speaker, buttons, frame, camera, etc. We selected differentiated Gaussian pulse excited lumped port, CPML boundary condition. We calculated return loss and impedance vs. frequency.

Table 2: GPU vs Intel Xeon CPU case1

	2 CPUs, 40 cores	4 GPUs
Simulation time	12:26:59	0:32:37
Time step	163548	163548
Mesh size	633 x 278 x 994 = 174,918,156	

Another case is simulated to benchmark the performance. This is a simple case, we employed increasingly fine mesh. We calculated return loss and impedance vs. frequency.

Table 3: GPU vs Intel Xeon CPU case2

	2 CPU, 40 cores	4 GPU
Simulation	9:51:59	0:31:7
Time step	55536	55536
Mesh size	1250 x 1250 x 40 = 62,500,000	

We also simulated the antenna case with different GPU cards. In figure 7, we illustrate the parallel efficiency.

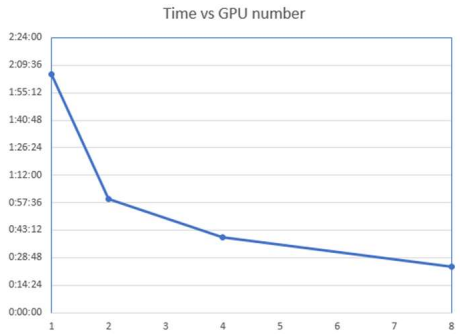

Figure 7: Parallel efficiency

6. Conclusions

In this article, we presented the implementation of GPU for FDTD method to accelerate the simulation time. We integrated it into our EM simulation software. If there is no available GPUs, we will disable GPU model and only use CPU model. We compared the performance between Nvidia GPU and Intel Xeon CPU. Modern GPU allows very efficient manipulation of large block of data. CUDA aware MPI helps to improve the parallel efficient between multiple GPUs.

References

[1] https://developer.nvidia.com/

[2] “CUDA C++ Programming Guide”, NVIDIA

[3] Wenhua Yu , Xiaoling Yang , Yongjun Liu, “Advanced FDTD Method”, Artech House, 2011