Implementation of the finite-difference method for solving Maxwell`s equations in MATLAB language on a GPU

N D Morunov[1]

[1]Samara National Research University, Moskovskoe Shosse 34, Samara, Russia, 443086

Abstract. The FDTD method of solving the Maxwell equations and methods for specifying a plane wave for it are considered. In addition, the features of implementation of calculations in MATLAB via graphics processing unit were considered. The comparative analysis of the difference solution of the Maxwell equations in the MATLAB language via the GPU for different ways of specifying a plane wave using an inexpensive graphics card is proposed. As a result, it is possible to speedup calculations to 60 times in the MATLAB language via the user video card NVIDIA GTX 960m.

1. Introduction

Numerical methods for solving the Maxwell equations require a large amount of time to solve simple problems of electrodynamics, not to mention complex ones. The most successful reduction in costs over time is parallel programming on GPUs and it is alsocalled general-purpose computing for graphics processing units (GPGPU). The well-known GPGPU programming tools are CUDA, OpenCL and computational shaders. Of these, CUDA is by far the most common and effective in the vast majority of cases [1].

It is necessary to know the C language to use CUDA for the calculations, which is more low-level language than MATLAB language. You have to deal with such difficulties as debugging of undefined behavior and memory leaks for programming mathematical problems in C language. In addition, not all scientists well-know the popular programming language C. The syntax of the MATLAB language is very simple, easy to learn, and is ideal for such scientists. There is a special package for parallel computing in MATLABcalled Parallel Computing Toolbox. It is possible to easily implement calculations on multi-core processors, on clusters and on GPUs via using this package and simple syntax.

The book A. Elsherbeni [2] described in detail the FDTD-method with implementation in the MATLAB language, but the author does not describe the implementation of parallel computations on a GPU in the same language. This was because at the time of writing the book MATLAB could not perform calculations on the graphics processing unit. This gap is filled with this article.

In the current article, we discuss in more detail the methods for specifying a plane wave in MATLAB using an inexpensive custom graphics card NVIDIA GTX 960m. Similar characteristics are GTX series 600-700 series for PC and 800m-900m series for laptops, the price of such a video card is low, about 200$.

2. Organization of parallel computations in MATLAB using a GPU

In additional materials to the book A. Taflove [3] there are scripts written by his assistant S. Hagness in the language MATLAB. There are only three of them: for one-dimensional, two-dimensional and three-dimensional cases of the FDTD method. Calculations in scripts are performed on the central processor. Consider the algorithm in MATLAB language for the three-dimensional case. At each step in time, the calculated values of the electric field components are written into the animation matrix, and then the animation is played using the movie function.

The algorithm of the program in the MATLAB language is presented below:

```
ex=zeros(ie,jb,kb);
…
hz=zeros(ie,je,kb);
for n=1:nmax
ex(1:ie,2:je,2:ke)=ca*ex(1:ie,2:je,2:ke)+...
cb*(hz(1:ie,2:je,2:ke)-hz(1:ie,1:je-1,2:ke)+...
hy(1:ie,2:je,1:ke-1)-hy(1:ie,2:je,2:ke));
…
ez(is,js,1:ke)=ez(is,js,1:ke)+...
srcconst*(n-ndelay)*exp(-((n-ndelay)^2/tau^2));
…
hz(1:ie,1:je,2:ke)=hz(1:ie,1:je,2:ke)+...
db*(ex(1:ie,2:jb,2:ke)-ex(1:ie,1:je,2:ke)+...
ey(1:ie,1:je,2:ke)-ey(2:ib,1:je,2:ke));
end;
```

Listing 1. Algorithm for the 3D case of FDTD method in MATLAB language.

In the MATLAB Parallel Computing Toolbox, there are three ways to implement parallel computations on a GPU: using a new type of gpuArray variable and various functions defined for this type; using arrayfun and bsxfun functions, which allow to define own function for processing GPU arrays; writing its CUDA-kernel and using it in calculations.

First of all, in order to remake the MATLAB code written for the CPU in the code written for the GPU, you need to convert all large arrays to the gpuArray type, you can do this either by explicitly casting the type to gpuArray, or by creating the array directly in the graphics card's memory (Listing 2). To transfer an array from the internal memory of the GPU to the RAM should be use the gather function. However, to display the array as a graph, this operation is not required. When drawing graphs, a set of vertices that define geometric primitives must be in the memory of the video card, so it is not meaningful to make data exchange.

```
ex=zeros(ie,jb,kb,'gpuArray');
…
hz= zeros(ie,je,kb,'gpuArray');
```

Listing 2. Declaring arrays for electric and magnetic fields directly in graphics memory.

Particularly useful functions are arrayfun and bsxfun, which very similar to the CUDA kernels in their work, the difference is the strict correspondence of the thread number to the indexes of the input arrays. The most effective way to implement parallel computations on a GPU in MATLAB is to reduce multiple synchronous operations over arrays to one function, where these operations are performed asynchronously, and using it in arrayfun or bsxfun. In addition, the bsxfun function is ideal for reduce large numbers of operations between small-sized vectors, for example, this technique is used to optimize the calculation of CPML.

```
ex(il:ir-1,jl:jr,kl) = bsxfun (@plus,ex(il:ir-1,jl:jr,kl), cb*hy_1D(il:ir-1));
ex(il:ir-1,jl:jr,kr+1) = bsxfun(@minus,ex(il:ir-1,jl:jr,kr+1), cb*hy_1D(il:ir-1));
```

Listing 3. The update of the electric field at the total field (TF) and the scattered field (SF) boundary using bsxfun.

```
Psi_eyx_xn(1:L,1:je,2:ke) = bsxfun (@times,Psi_eyx_xn(1:L,1:je,2:ke),cpml_b_en(1:L))+...
bsxfun (@times,hz(1:L,1:je,2:ke)-hz(2:L+1,1:je,2:ke),cpml_a_en(1:L));
ey(2:L+1,1:je,2:ke) = ey(2:L+1,1:je,2:ke) + cb_psi(1:L,1:je,2:ke).*Psi_eyx_xn(1:L,1:je,2:ke);
Psi_ezx_xn(1:L,2:je,1:ke) = bsxfun (@times,Psi_ezx_xn(1:L,2:je,1:ke),cpml_b_en(1:L))+...
bsxfun (@times,hy(2:L+1,2:je,1:ke)-hy(1:L,2:je,1:ke),cpml_a_en(1:L));
ez(2:L+1,2:je,1:ke) = ez(2:L+1,2:je,1:ke) + cb_psi(1:L,2:je,1:ke).*Psi_ezx_xn(1:L,2:je,1:ke);
```

Listing 4. Updating Ex CPML components using bsxfun.

MATLAB supports a set of functions for working with CUDA directly: loading the kernel, changing kernel parameters, etc. This method involves writing a kernel in C (the actual procedure that will be performed on the GPU) and connecting it to the MATLAB application. Next, you need to configure the loaded kernel: specify the number of maximum possible parallel threads; specify the number of blocks, and execute the kernel procedure.

The third method though is the best, but the task of the article was to implement the finite-difference solution using MATLAB language. Moreover, it is necessary to write CUDA kernels in the C language before connectthem to MATLAB.

MATLAB is more suitable for vector calculations, so whole code should be vectorized at first. When working with Parallel Computing Toolbox, another one feature was noticed: if a size of the vector is small (about 10 elements), then this vector should not be transferred to the graphics card's memory, since in some cases this will only slow down the work. Moreover, if over a large number of iterations in the algorithm, only one element of the vector is used, and its values doesn`t change during these iterations, it is worthwhile to cut this value from the vector and transfer it to the local work area.

MATLAB uses memory both for storing raw data and result data, and for storing intermediate data used in calculations. Thus, the maximum number of nodes can be defined as:

$$I_{\max} = \frac{V_{\max}}{v(a_{st} + a_{comp})},$$

where $V_{\max}$ – available memory capacity; v –amount of data consumed by a single node value; a_{st} – number of arrays using by store data; a_{comp} – number of arrays using by compute data.

Graphical and operative memory are allocated in different ways in the same cases. For example, copies of arrays are created with using RAM, whenever there is a possibility to use the same memory cell two or more times for calculating one expression, i.e. the presence of conflicts when accessing memory. Therefore, the array is duplicated when using indexes and when using non-element operators (*, ^). In the case of graphics memory, arrays are duplicated usually for each individual argument and the return value in the expression. In addition, there is one general rule that can avoid copying an array: if the return value is contained as an argument in the expression, then this argument is not copied.

For example, consider the following expression written in MATLAB language:

A = ca.*A + cb.*(B(2:end)-B(1:end-1)).

We will assume that the expression is written correctly and the sizes of all arrays used as arguments coincide. If all the arrays are in RAM, then in the process of calculations the program in MATLAB language will copy array B two times. If they are in graphics memory, then they will copy 5 arrays. As a result, the graphics memory will need almost 1.5 times more.

When performing parallel computations on a GPU, MATLAB automatically optimizes the calculations in the cycle, allocating more graphics memory. If the memory enough for arrays copies for all expressions in the loop, then MATLAB additionally parallelizes the calculation of the expressions. In this case, you can set an effective amount of graphics memory and the maximum amount of graphics memory.

In accordance with all the above listed features of the Parallel Computing Toolbox, the algorithm of C. Hagness (Listing 1) was adapted to run on the GPU.

3. Experimental research of FDTD implementation in MATLAB language

An experiment was conducted to investigate the effectiveness of the implementation of the 3D case of the FDTD method.

In the experiment were used an NVIDIA GTX 960m graphics card with GDDR5-2Gb internal memory and a number of CUDA-compatible shader processors - 640 with a base clock speed of 1.1 GHz. Moreover, a 4-core Intel Core i5-6300HQ CPU with a base clock speed of 2.3 GHz and a maximum frequency of 3.2 GHz, with a cache size of 6 MB. Also on the laptop was installed 8GB of RAM type DDR4-2133 andnorth bridge - Intel Skylake-H IMC.

The next step was to develop a other script for profiling the work of the computational process on different processors. Based on the algorithms considered earlier, Functions were written to perform calculations on the CPU and on the GPU. The input of this function is given the size of the grid area and the number of iterations, and the output is the execution time of the calculations.

Consider the profiler code in a simpler form:

```
c_min = 30;c_max = 250;dc = 10;nmax=6;
T = (c_max-c_min)/dc+1;
cells = c_min:dc:c_max;
fori = 1:T
    cpu_time(i) = fdtd3D_cpu(cells(i), nmax);
    gpu_time(i) = fdtd3D_gpu(cells(i),nmax);
end;
speedup = cpu_time./gpu_time;
cells_axis = cells.^3 / 10^6;
cpu_perfomance_axis = cells.^3 ./ cpu_time / 10^6;
gpu_perfomance_axis = cells.^3 ./ gpu_time / 10^6;
figure;
subplot(3,1,1);
plot(cells_axis, cpu_perfomance_axis);
subplot(3,1,2);
plot(cells_axis, gpu_perfomance_axis);
subplot(3,1,3);
plot(cells_axis, speedup);
```

Listing 5. Profiler code in the MATLAB language.

Four combinations of initial and boundary conditions were considered and for each of them two functions fdtd3D_cpu and fdtd3D_gpu were implemented (Listing 5). A source that generated a plane wave in two ways, TFSF and TFRF [4] was given as initial conditions. Perfect electrical conductor (PEC) boundary conditions is defined along the axis of propagation and cyclic boundary conditions are given for other directions for two cases with the TFRF method.Perfect electric conductor (PEC) boundary conditions was defined in the other two cases with the TFSF method. In addition, Convolutional Perfectly Matched Layer (CPML) [2] was defined for each method of specifying a wave.

As a result, of the profiling script execution, plots were obtained. Plots show the dependence between calculation rate and number of cells. Calculation rate is a millions of processed cells per second. Figure 1 shows the characteristics of a computational process that calculates the propagation of a plane wave in a computational domain bounded by a perfect electric conductor boundary without the use of CPML, and in Figure 2 with CPML. As a result, the plot with dependence between speedup and number of cells for each case was calculated and presented in Fig. 3.

There is sharp decrease in the calculation rate on the plots (Figures 1b and 2b) for GPU after some value of the grid size. This is due to the lack of video memory on the graphics card, and the value corresponds to effective memory. On the plots of dependence between CPU calculation rate and the number of nodes (Figures 1a and 2a), a sharp jump is seen at the very beginning. This is because the amount of data necessary for the computing task is placed in the central processing unit’s cache and the access speed to the memory is high. The amount of data required for the computational task increases and number of cache misses increases too. Moreover, speed of access to memory decreases, as well as the performance of computations.

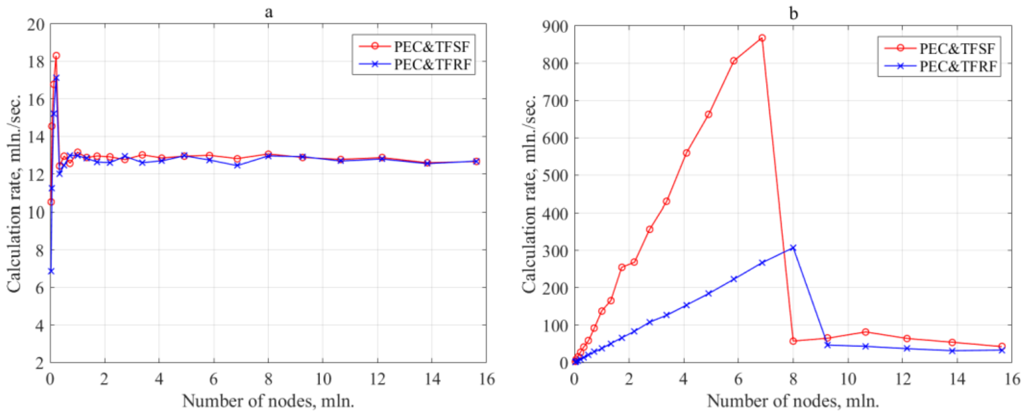

Figure 1. Plots of dependence between calculation rate (million nodes / sec.) and the number of processed nodes (million) withoutCPML: a) on the CPU; b) on the GPU.

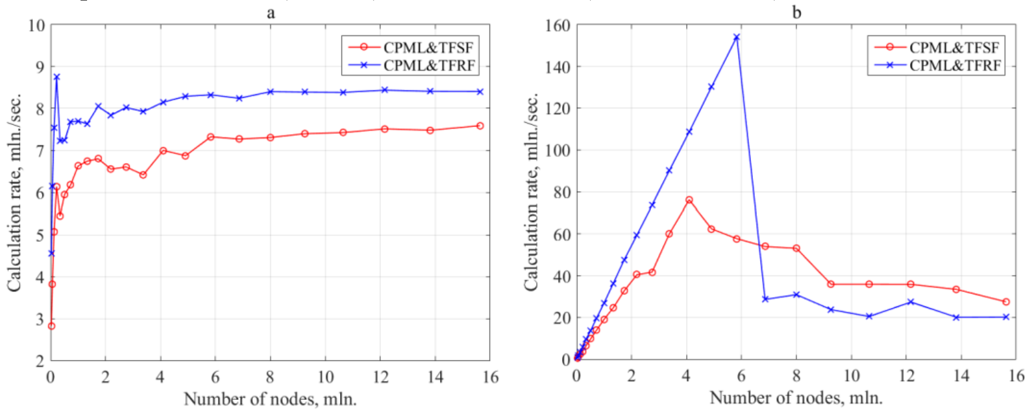

Figure 2. Plots of dependence between calculation rate (million nodes / sec.) and the number of processed nodes (million) with CPML: a) on the CPU; b) on the GPU.

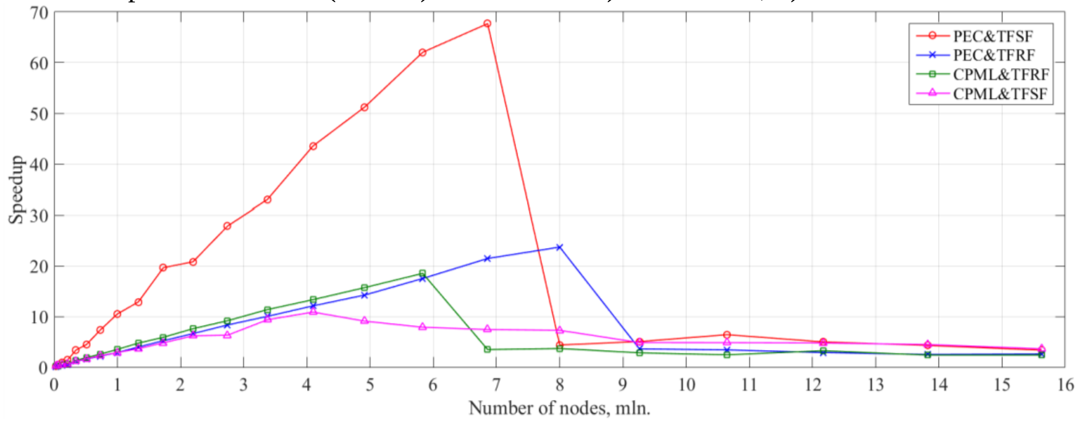

Figure 3. Plots of dependence between speedup and the number of processed nodes (million).

The calculation rate increases well when using the GPU in both cases with TFRF (Figures 1 and 2). However, there is a calculation rate decline when using CPML in cases with TFSF. This behavior can be explained by the following: the boundary conditions in TFSF (an ideal electrical wall) are set easily without adding extra vectors, but CPML is set on each of the 6 faces; TFRF is given periodic boundary conditions, which increases the number of vectors, and CPML is only set on 2 faces. The use of CPML in calculations creates a greater number of small vectors, larger the dimension of the field to which these perfect matched layer is set. Accordingly, the volume of parallel computations decreases because of the large number of individual vectors.

4. Conclusion

Let's sum up the analysis. It should be noted that if it is required to solve the computational problem of electrodynamics, where a plane wave is used as a source, it is preferable to use the TFRF method. In this paper we obtained the results (Figure 3), based on which it can be stated that it is possible to perform computational tasks of electrodynamics on inexpensive video cards with speedup 20 times using CPML and 60 times without it.However, another problem arises: the limited amount of graphics memory in comparison with the operating memory. Solve this problem is decomposing, splitting the range of values into blocks of such a volume, in which the computational process has the maximum acceleration.

5. References

[1] Golovashkin D L and Kasanskiy N L 2011 Solving diffractive optics problem using graphics processing units *Optical Memory and Neural Networks (Information Optics)* **20(2)** 85-89

[2] Elsherbeni A Z and Demir V 2008 *The finite-difference time-domain method for electromagnetics with MATLAB simulations* (Raleigh: SciTech Publishing) p 426

[3] Taflove A 1995 *Computational Electrodynamics: the finite-difference time-domain method* (London: Artech House) p 599

[4] Soifer V A 2014 *Diffractive nanophotonics* (CRC Press) p 704