摘 要

本文针对时域有限差分（Finite Difference Time Domain，FDTD）方法的两种主流硬件并行加速途径进行了研究。针对使用矢量处理器（Vector Algorithm Logic Unit，VALU）的加速方法，首先进行了加速的理论分析，然后基于现有方案，同时结合FDTD的边界条件的特征，提出了一种新的使用VALU加速计算的计算模型。经过实验，相比以往方案改计算模型可以减少约3.45%的计算时间。针对使用统一计算设备架构（Compute Unified Device Architecture，CUDA）的图像处理器（Graphic Processor Unit，GPU）的加速方法，我们实现了对截断边界、Mur吸收边界以及CPML吸收边界的加速。对比两种硬件并行加速方案，我们可以看到CUDA有更优越的表现。

关键词：时域有限差分，CUDA，矢量处理器，数据并行，硬件加速

ABSTRACT

In this paper, we did some researches on two main hradware accelerating methods. As to the way of using vector algorithm logic unit (VALU), we analyze its theory at first, and then we proposed a new computational model based on traditional computational model under some specific boundary conditions. Compared with traditional model, the advanced model can save about 3.45% time.

The another method, accelerating by compute unified device arthitecture (CUDA), we implement FDTD under PEC, Mur, and CPML boundary conditions. Then, we compared this way with VALU way. At last, we obtain the performance of every method.

Keywords: FDTD, CUDA, VALU, data parallelism, hardware accelerate

Contents

Chapter 1 Introduction

1.1 Background

The theory and application of electromagnetic wave has been developed greatly in last about one hundred years since the Maxwell's equations were established. The researches and applications in the area of electromagnetic wave have dived into every subarea, like electromagnetic scattering, electromagnetic radiation, modeling of waveguides, electromagnetic imaging, and electromagnetic probe. In real environment, the process of propagation of electromagnetic wave is very complicated, like the scattering of an complex object, the real communication in city at a complicated topography, and the propagation in waveguide. So, it is very useful if we can know the specific feature of electromagnetic wave under some real environments like we mentioned before and we already have two important ways to do it, which are simulating and theoretical analyzing. The theoretical analyzing method, however, can only answer some typical problems, not too complex problems derived from real environments and involving real electromagnetic arguments. Unfortunately, we always have to deal with the later one, which means that we have to use the simulating way to solve those problems derived from real circumstances, and motivated the development of computational electromagnectics. So far, there are many computational numerical techniques to overcome the inability of analytically calculation to derive closed solutions of Maxwell's equation, like Methos of Moments (MoM), Boundary Element Method (BEM), Fast Multipole Method (FMM), Finite Element Method (FEM), and Finite Difference Time Domain (FDTD).

In 1996, K. S. Yee [**?**] present the FDTD scheme in his seminal paper. This method discretiz time-dependent Maxwell's equations by applying centered finite difference operators on staggered grids in space and time for each electric and magnetic vector field component in Maxwell's equations. Then the resulting finite difference equations are solved in a leapfrog manner: calculating the electric field vector components in a volume of space at a given instant; then do the same thing to the magnetic field vector component in the same spatial area at the next instant in time; repeating the process over and over again until the desired transient. FDTD has many strengths, so it became the primary means to solve those problems and got a wide range of applications like radar signature technolofy, antennas, wireless communications devices, digital interconnections, even to photonic crystals, solitons, and biophotonics.

There are two main problems laying on the ways of the development of FDTD's application. The first is boundary condition. As the power of computer is finite, we can only so computation to simulate the electromagnetic wave in a finite volume space. So, if we want to simulate the propagation of electromagnetic wave for open-region FDTD, we have to give out an appropriate boundary condition. To solve this problem, there are many solutions. In 1969, Taylor[**?**] et al. presented a extrapolation boundary condition. In 1981, Mur[**?**] proposed a absorbing boundary condition (ABC). Berenger [**?**] developed the perfectly match layer (PML) in a seminal 1994 paper. Based on the original work of Berenger, various PML formulation were proposed. One of the most famous modified and extended implementation is uniaxial PML (UPML), which was proposed by Sacks et al. [**?**] and Gedney et al. [**?**]. Another famous modified PML is convolutional PML (CPML), which was proposed by Chew and Weedon et al. [**?**]. The Mur, UPML, and CPML are most commonly used ABCs. The Mur ABC is the most simple one among them, and the latter two PML ABCs have more strong ability to absorb evanescent waves.

1.2 The Topics Covered in This Paper

To implement the FDTD, we need to establish grids in the computational domain at first. Concerned the Courant condition, the grid spatial discretization must be fine enough to resolve the smallest wavelength, which means the number of grids in the computational domain has a low boundary. Hence, we have a very big computation task as the significant number of grids as we need to compute every grid at a given instant. So, objects with long and thin feature are not suitable to model by using FDTD. As the size of grid, the time step need to fit into the Courant condition, too. So, we have to wait a long time if the desired transient is far form the initial time. That is the second problem of FDTD: it always take too much time to solve a problem.

Parallel processing is the main answer to solve the second problem. FDTD have a natural character of parallelism, that is, the FDTD method requires only exchanging field on the interface between adjacent sub-domains. According to this character, some parallel FDTD algorithm[? ? ? ?] has been developed. In those parallel method, the original computational domain has been divided into some sub-domains and assign each sub-domain to a individual computational core. The computational core complete its own work independently at any give instant, and then exchange the data of sub-domain's boundary with its adjacent computational core before the next given instant. One famous example of this method is message passing interface (MPI).

The MPI method, and other method who adopt the way dividing original computational domain belong to task-level parallelism (TLP), which involves the decomposition of a task into sub-tasks and then complete these sub-tasks simultaneously and sometimes cooperatively. The instruction-level parallelism (ILP), which is lower than TLP, can be achieved by all modern CPU automatically without human's intervention. The lowest level parallelism, data-level parallelism (DLP), still has some potential power of computation of FDTD.

1.2.1 The Utilizing of Vector Processor

At first, people use CPU directly to compute. In this way, the computations are completed by arithmetic logic unit (ALU), which can only compute one data at once. To achieve data parallelism, we use vector processor (VP) instead of ALU to compute. A vector processor is a central processing unit that can operate on one-dimensional arrays of data which called vectors by using instructions. Vector processor improves much about the performance of computation in some areas, especially in numerical simulation tasks. Now, the instruction set to operate vector is provided by almost commercial CPUs, like Intel or AMD, provided VIS, MX, SSE, and AVX, etc. Therefore one of the advantages of VP is easy to access.

In past researches about accelerating the FDTD method based on data parallelism. L. Zhang and W. Yu [?] used SSE to accelerate 3D FDTD method with single precision floating point arithmetic. M. Livesey, F. Costen, and X. Yang [?] extended the work to doubleprecision floating arithmetic. The way to use vector processor and the pseudocode were presented by Y. L et al. [?]. However, all of those researches ignored the differences of the characters between different boundary conditions remain some potencies to be exploit. We found that the computational domain could be modified to enhance the efficiency of computation for some specific boundary conditions, like MUR. Considering the characters of some ABCs, like Mur ABC, we give out a new computational model and its implementation developed by C++ under 2D FDTD condition as an example and analyze the performance by using Visual Studio. This model could be extended to 3D FDTD method.

1.2.2 The Utilizing of CUDA

Though in every parallelism level we have schemes to accelerate the FDTD, the CPU is designed to deal with every possible complex general operation by higher operation frequency, more registers, and more advanced ALU, not to solve big data computation task professionally. So, it cannot make full use of CPU and meet the need of efficiency of FDTD if we use CPU to execute FDTD. If we concentrate our attention on graphic processor unit (GPU), we will find that GPU, which has hundreds of computational core, and was designed to deal with big volume data and repetitive computation, is a better choice to execute FDTD.

The concept of GPU was proposed in 1999. Sean E. Krakiwsky et al.[**?**] tried to use GPU to accelerate FDTD at the first time in 2004. At that time, the GPUs were not designed to general purpose computation, so it was difficult to use GPU to complete FDTD as people who want to use it need to understand the architecture of it and a special hardware language. Things got better in 2006, as GPU had already had general purpose computational architecture at that time. In 2007, the computational unified device architecture (CUDA) was published, which is a milestone. The presence of CUDA allowed people use the programming languages they have already learned, like Fortran, C, or C++, which speed up the research about using GPU on FDTD. J. A. Roden and S. Gedney[**?**] proposed a implementation of FDTD under the CPML ABC. However, they did not provide any details. Veysel Demir [**?**] proposed a implementation of FDTD under periodic boundary condition (PBC). In this paper, we use CUDA to accelerate 2D FDTD under Mur ABC to evaluate the performance to CUDA.

Chapter 2 The Theory of FDTD

2.1 Yee Cell

Maxwell's equations are a set of equations which can be written in differential form or integral form. They are the foundation of macroscopic electromagnetic phenomenas. There are two kinds of numerical solver to Maxwell's equation. One kind of solvers were developed from integral form of Maxwell's equations, called integral equation solvers, including MoM, BEM etc. Another kind of solvers, like FDTD and FEM, were developed from differential form of Maxwell's equations, called differential equation solvers.

FDTD is based on Maxwell's equations in differential form, which are shown as follows. Then Maxwell's equations are modified by discretized in central-difference way.

$$\nabla \times H = \frac{\partial D}{\partial t} + J, \tag{2-1}$$

$$\nabla \times E = -\frac{\partial B}{\partial t} - J_m. \tag{2-2}$$

In Cartesian coordinate system，the equation(2-1) and (2-1) are written in following forms:

$$\begin{cases} \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} = \varepsilon \frac{\partial E_x}{\partial t} + \sigma E_x \\ \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} = \varepsilon \frac{\partial E_y}{\partial t} + \sigma E_y \\ \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} = \varepsilon \frac{\partial E_z}{\partial t} + \sigma E_z \end{cases} \tag{2-3}$$

$$\begin{cases} \frac{\partial E_z}{\partial y} - \frac{\partial H_y}{\partial z} = -\mu \frac{\partial H_x}{\partial t} - \sigma_m H_x \\ \frac{\partial E_x}{\partial z} - \frac{\partial H_z}{\partial x} = -\mu \frac{\partial H_y}{\partial t} - \sigma_m H_y \\ \frac{\partial E_y}{\partial x} - \frac{\partial H_x}{\partial y} = -\mu \frac{\partial H_z}{\partial t} - \sigma_m H_z \end{cases} . \tag{2-4}$$

The six equations in (2-3) and (2-4) are a set of partial differential equations in space-time formulations form which represent each filed vector component. They are hard to deal in that form, so we need to discretize them in space and time at first.Let $u(x, y, z, t)$ represent any field vector component of E or H in Cartesian coordinate system. On the aspect of space, we assume that the discreteness on space is uniform, which means the space steps, also called the lengths of grids, are equal to each other in x, y, and z directions and written as Δx, Δy, and Δz respectively. We also use i, j, and k to represent the grid index in directions of x, y, and z respectively. On the aspect of time, wo assume the discreteness is uniform, too. Besides, we adopt the symbol Δt to represent the length of time step, which also called the distance between two iterations, and n to represent the index of time steps. After all, we can represent any field vector component in the following notion to indicate the location where the field vector components are sampled in space and time:

$$u(x,y,z,t) = u(i\Delta x, j\Delta y, k\Delta z, n\Delta t) = u^n(i,j,k). \tag{2-5}$$

There are three forms of finite difference, forward, backward, and central difference, which are considered commonly. Here we pick the central difference as it has second-order numerical accuracy. Let us take the x direction as an example to illustrate a field vector component's spatial first partial derivative:

$$\frac{\partial u^n(i,j,k)}{\partial x} \approx \frac{u^n(i+\frac{1}{2},j,k) - u^n(i-\frac{1}{2},j,k)}{\Delta x}. \tag{2-6}$$

And the its first partial derivative on time is:

$$\frac{\partial u^n(i,j,k)}{\partial t} \approx \frac{u^{n+\frac{1}{2}}(i,j,k) - u^{n-\frac{1}{2}}(i,j,k)}{\Delta x}. \tag{2-7}$$

Now we have discretized the six equations in (2-4) and (2-3). The next step we should do is considering how we position those discrete points of every field vector component. The answer is Yee cell.

In space, we position those discrete points like what illustrated in 2-1. This is the spatial structure of Yee cell.

We can see from the figure 2-1 that in Yee cell each three components of E and H are discretized on the surface. Electric field components E_x, E_y, and E_z are on the center of edges, and magnetic field components H_x, H_y, and H_z are on the center of surfaces. In this way, for each electric discrete point there are four magnetic discrete points encircling it. Conversely, for each magnetic discrete point there will be four electric discrete points encircling it. This way of distributing discrete points fit into Faraday's law and Ampere's law in nature. In time aspect, Yee cell adopted the time-stepping manner. For any discrete point in space, the updated value of the electric filed in time is dependent on the stored

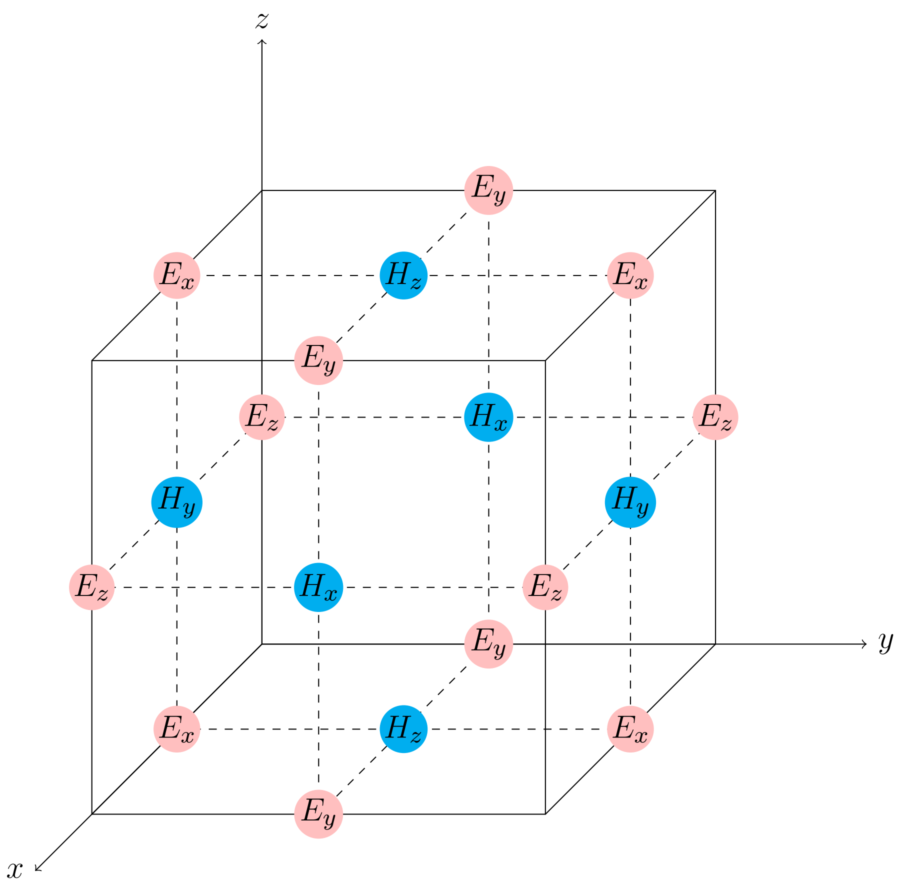


图 2-1 The spatial discrete structure of Yee cell value of magnetic field of the last time. Iterating the process in a marching-in-time way, we can make an analog to the continuous electromagnetic waves's propagation.

In summary, Yee cell can describe the interaction between electrics and magnetics naturally. Given a specific problem with specific coefficients, initial state, and boundary condition, we can use FDTD to obtain the distribution of electromagnetic waves in any given transient.

To indicate the location of discrete points in space and time, we need to give any location of Yee cell a serial number. Here, we follow this rule: in any direction, the serial number of Yee cell's edges is integer, and the serial number of the center of Yee cell's edge is half integer; all time location of electric discrete points are integer, and all time location of magnetic discrete points are half integer. For each field vector component, the serial number in space is illustrated in the following table.

2.2 The Updating Formulations

Let us take the (2-3) as an example to explain the how to update a field vector component and the updating formulations. According to the (2-6) and (2-7), we change (2-4) into the discrete form, which shown below:

$$
\begin{aligned}
E_x^{n+1}\left(i+\frac{1}{2},j,k\right) =& CA(m)\cdot E_x^{n}\left(i+\frac{1}{2},j,k\right)+CB(m) \\
&\cdot\left[\frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y}\right. \\
&\left.-\frac{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z}\right].
\end{aligned}
\tag{2-8}
$$

And there has

$$
CA(m)=\frac{1-\frac{\sigma(m)\Delta t}{2\varepsilon(m)}}{1-\frac{\sigma(m)\Delta t}{2\varepsilon(m)}},
\tag{2-9}
$$

and

$$
CB(m)=\frac{\frac{\Delta t}{\varepsilon(m)}}{1+\frac{\sigma(m)\Delta t}{2\varepsilon(m)}}.
\tag{2-10}
$$

表 2-1 The serial number of components of *E* and *H* in Yee cell

Field components	The location in space x	y	z	The location in time
E_x	$i+\frac{1}{2}$	j	k	
E_y	i	$j+\frac{1}{2}$	k	n
E_z	i	j	$k+\frac{1}{2}$	
H_x	i	$j+\frac{1}{2}$	$k+\frac{1}{2}$	
H_y	$i+\frac{1}{2}$	j	$k+\frac{1}{2}$	$n+\frac{1}{2}$
H_z	$i+\frac{1}{2}$	$j+\frac{1}{2}$	k	

We use m to represent the spatial location the discrete point have which is at the right of equal sign.

To explain the algorithm in a clear way, we assume all mediums are lossless medium, ie $\sigma = \sigma(m) = 0$. Hence, there has $CA(m) = 1$, and $CB(m) = \frac{\Delta t}{\varepsilon(m)}$. So, now, (2-8) is

$$\begin{aligned} E_x^{n+1}\left(i+\frac{1}{2},j,k\right) =& E_x^n\left(i+\frac{1}{2},j,k\right)+\frac{\Delta t}{\varepsilon(m)} \\ &\cdot\left[\frac{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y}\right. \\ &\left.-\frac{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z}\right]. \end{aligned} \tag{2-11}$$

In the same way we modified the field vector components of *H*. Let us take the H_x as an example, now the first equation in (2-4) is:

$$\begin{aligned} H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right) =& H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)+\frac{\Delta t}{\mu} \\ &\cdot\left[\frac{E_y^n\left(i,j+\frac{1}{2},k+1\right)-E_y^n\left(i,j+\frac{1}{2},k\right)}{\Delta z}\right. \\ &\left.+\frac{E_z^n\left(i,j,k+\frac{1}{2}\right)-E_z^n\left(i,j+1,k+\frac{1}{2}\right)}{\Delta z}\right] \end{aligned} \tag{2-12}$$

All other field vector components can be processed in the exactly same way we stated above. Therefore, we obtain each field vector component's discrete form updating formulation.

2.3 Courant Condition

According to the last section, we know that instead of a set of continuous partial differential equations, FDTD solving a set of discretized Maxwell's equations. So, like other numerical approximate methods, FDTD has a necessary condition for convergence while solving partial differential equations numerically. As a consequence, the time step must be less than a certain time.

At first, we consider the time step, Δt. As any field can be deposited into several harmonic electromagnetic fields, we examine the limitation of Δt in harmonic electromagnetic field.

Here is a harmonic electromagnetic field:

$$u(x,y,z,t) = u_0 exp(j\omega t). \tag{2-13}$$

The first-order partial differential equation of (**??**) respect to time is:

$$\frac{\partial u}{\partial t} = j\omega u. \tag{2-14}$$

By using central difference approximations to the left side of equation (2-14), we obtain:

$$\frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} = j\omega u^n, \tag{2-15}$$

In which $u_n = u(x,y,z,n\Delta t)$.

Let the growth factor q be:

$$q = \frac{u^{n+\frac{1}{2}}}{u^n} = \frac{u^n}{u^{n-\frac{1}{2}}}. \tag{2-16}$$

Then, combining the equation (2-16) and (2-15), we obtain this equation:

$$q^2 - j\omega\Delta t q - 1 = 0 \tag{2-17}$$

Solving the equation (2.3), we will have:

$$q = \frac{j\omega\Delta t}{2} \pm \sqrt{1 - \left(\frac{\omega\Delta t}{2}\right)^2}. \tag{2-18}$$

If we want the value of fields converge along the marching of time, the condition $|q| \leqslant 1$ must be satisfied. Hence, there is

$$\frac{\omega\Delta t}{2} \leqslant 1. \tag{2-19}$$

Equation (2-19) is the necessary condition of Δt required by a field which need to be stable.

For Maxwell's equations, which have six field components, we know that all rectangular components of electromagnetic field fit in the homogeneous wave equation which is following:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} + \frac{\omega^2}{c^2} f = 0. \tag{2-20}$$

As any wave can be expanded to plain waves, so we consider the solution of plain waves of homogeneous wave equation:

$$u(x,y,z,t) = u_0 exp[-j(k_x x + k_y y + k_z z - \omega t)]. \tag{2-21}$$

Substituting the equation (2-21) into (2-20), and discretize the result by using central difference, then we obtain the following equations:

$$\frac{\sin^2\left(\frac{k_x\Delta x}{2}\right)}{\left(\frac{\Delta x}{2}\right)^2} + \frac{\sin^2\left(\frac{k_y\Delta y}{2}\right)}{\left(\frac{\Delta y}{2}\right)^2} + \frac{\sin^2\left(\frac{k_z\Delta z}{2}\right)}{\left(\frac{\Delta z}{2}\right)^2} - \frac{\omega^2}{c^2} = 0. \tag{2-22}$$

The c is the speed of light in media among it.

Simplify the equation (2-22) and substitute it into (2-19). Then we obtain

$$\left(\frac{c\Delta t}{2}\right)^2 \left[\frac{\sin^2\left(\frac{k_x\Delta x}{2}\right)}{\left(\frac{\Delta x}{2}\right)^2} + \frac{\sin^2\left(\frac{k_y\Delta y}{2}\right)}{\left(\frac{\Delta y}{2}\right)^2} + \frac{\sin^2\left(\frac{k_z\Delta z}{2}\right)}{\left(\frac{\Delta z}{2}\right)^2}\right] = \left(\frac{\omega\Delta t}{2}\right)^2 \leqslant 1. \tag{2-23}$$

This equation is true under the following condition:

$$c\Delta t \leqslant \frac{1}{\sqrt{\frac{1}{(\Delta x)^2}\frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}}}. \tag{2-24}$$

The equation (2-24) give out the relationship between time step Δt and space step Δx, which called courant condition.

2.4 The limitation of Δx

In the last section, the relationship between the time step and the space step was been determined. Taking the 1D situation as an instance, from the equation (2-22) we have:

$$\frac{\sin^2\left(\frac{k_x \Delta x}{2}\right)}{\left(\frac{\Delta x}{2}\right)^2} - \frac{\omega^2}{c^2} = 0. \tag{2-25}$$

We can observe the fact from the equation (2.4) that the dispersion can be avoided only if $\Delta x \to 0$. So, we need to evaluate to what extent the Δx can be treated as closed to 0 in numerical approximation. According to triangle functions, when $\phi \leqslant \pi/12$, $sin\phi \approx \phi$. So, there has

$$\Delta x \leqslant \frac{2\pi}{12k} = \frac{\lambda}{12}. \tag{2-26}$$

Equation (2-26) is the requirement for space step Δx. For circumstances of 2D or 3D, they are the same as 1D situation, just make space steps of each dimension meet the condition (2-26). To signals whose band are wide and belong to non-homogeneous waves, make sure the space step meet the requirement (2-26) of the shortest wave length.

2.5 Boundary conditions of FDTD

In theory, by following the rules of FDTD we can simulate almost every electromagnetic wave in the infinite space and the length of time is infinite, too. However, as the power of computation is limited, and problems are always have significant big size, we can, and should simulate the the waves in the target area we concerned even in fact those waves are propagating in the infinite space. One possible way to do that is to design boundary conditions properly. Given a problem, we can compute discrete field points which are in the target area to observe how the wave changes in the whole area. There are two kinds of boundary conditions, absorbing boundary conditions (ABC) and truncating boundary conditions (TBC). In reality, people always adopt ABC, which allow them simulate the situation a wave propagating in the infinite space. Among ABCs, Mur and perfectly matched layer (PML) are two main boundary conditions.

2.6 The ways of parallel computing of FDTD

To satisfy the Courant condition, the grids of Yee cell must be fine enough, which means the number of discrete field points we need to compute will be significantly massive, therefore, solving a real problem in a big size seems not practical. Parallel computing, as to this problem, is a quite efficient solution.

There are three level of parallelism. The top level is task level. In this level, a huge task need to be completed is divided into several independent smaller tasks. After those smaller tasks computed by some computing cores, we merge the solutions of those smaller tasks into a integral solution, which answers the original task. According to the parallelism nature of FDTD, that all computing cores need exchange some boundary data with its adjacent computing cores, FDTD is suitable to be computed in parallel computing. So far, we have massage passing interface (MPI) method and parallel virtual machine (PVM) method of this parallelism level.

The lower level of parallelism is instruction level parallelism. This level of parallelism always considered by CPU manufacturers, hardly by users.

The lowest level of parallelism if data level. In this level, several data can be evaluated by one instruction simultaneously if a task have been reorganized appropriately. In FDTD, as all discrete field points of the same field vector component have a shared updating formulation, we can compute several discrete field points by every single instruction when those points belong to the same field vector component. For parallelism in this level, there are some implementations of FDTD by using vector processor instruction sets.

2.7 Conclusion

In this chapter, de discussed several aspects of FDTD. First, we introduce the theories of FDTD algorithm. Then we describe the updating formulations for every field vector components, which are the key of FDTD. After that, the necessary condition, Courant condition was stated. Satisfying this condition make sure all field vector components keep stable and being convergent, which make FDTD useful in reality. Then we discussed the boundary conditions of FDTD, which influence the accuracy of FDTD. Finally, we introduced several ways of parallel computing to make FDTD faster. All things we discussed above in this chapter are the foundations of the works in this paper.

Chapter 3 Accelerating FDTD by Using VP

In this chapter, we use the simulation of 2D TM mode as an instance to explain the scheme of accelerating FDTD by using vector processor.

3.1 Data Parallelism and VP

The vector operations, which means processing several data with one operation simultaneously, like inner production, are always used by people. Note that A one-dimensional array of numbers that packed in vector processor is called as a vector, against the scalar which includes only one number, not what is fully described by both a magnitude and a direction in physics. The terms vector and scalar address only the number of data. There are scalar processor and vector processor in CPU, which illustrated in Fig. 3-1. Through the Fig. 3-1, we can see that 4 numbers can be operated at the same time in a vector processor while just one number can be processed in scalar processor.

VP is a central processor unit that can operate vectors by a set of specific instructions. Several data are packed in some vectors by using instructions, and sent to VP. After being processed with desired operations, those vectors are sent to memory.

However, there is one important point we need to address that data object in memory should be aligned by 16 bytes when we creating them to increase the efficiency of data loads and stores to from the processor. So, we need to allocate memory suitably aligned in 16 bytes other other number of bytes designed by CPU manufacturer at the beginning of a program. If 4 contiguous float data without the first one aligned by 16 bytes, they could also be loaded into the processor, but they still should continuous in memory, and it will be slower by 4.5 times than aligned situation.

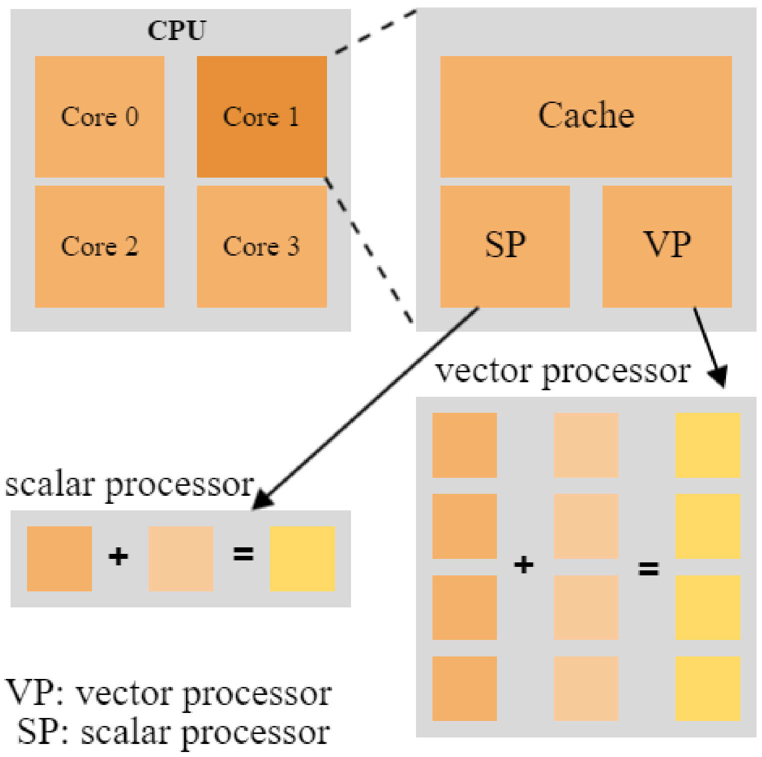


图 3-1 Vector processor and scalar processor

3.2 The Scheme of Accelerating FDTD by Using VP

Taking TM mode as example, the discrete form of updating formulations of the field vector components H_x, H_y, and E_z are as following:

$$H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2}\right)=H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2}\right)+\frac{\Delta t}{\mu}\left(\frac{E_z^n(i,j)-E_z^n(i,j+1)}{\Delta y}\right) \tag{3-1}$$

$$H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j\right)=H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2},j\right)+\frac{\Delta t}{\mu}\left(\frac{E_z^n(i+1,j)-E_z^n(i,j)}{\Delta x}\right) \tag{3-2}$$

$$\begin{aligned}E_z^{n+1}(i,j)=&E_z^n(i,j)\\&+\frac{\Delta t}{\epsilon}\left(\frac{H_y^{n+1/2}(i+\frac{1}{2},j)-H_y^{n+1/2}(i-\frac{1}{2},j)}{\kappa_x\Delta x}\right.\\&\left.-\frac{H_x^{n+1/2}(i,j+\frac{1}{2})-H_x^{n+1/2}(i,j-\frac{1}{2})}{\kappa_y\Delta y}\right)\end{aligned} \tag{3-3}$$

The next, we take a 3×3 size grids as an instance to explain the traditional data parallelism scheme intuitively. According to the way of Fig. 2-1, we position all field vector components as what shown in the Fig. 3-2 in the 3×3 size Yee cells. The serial numbers in grids represent the position of the corresponding discrete field point, which the position in y direction, ie the row it is on, is represented by the first number of a serial number, and correspondingly, the position in x deirection, ie the column it is on, is represented by the second number of a serial number.

The traditional data parallelism method of FDTD, did some modifications, which is adding some auxiliary H_x field points at the boundary of x direction, to the spatial organization shown in Fig. 3-3. Now, let N_x and N_y are the number of Yee cell of x, y direction respectively. Therefore, the number of discrete field points of each field vector component are shown as Table 3-1.

表 3-1 The number of discrete points of each field component in each direction

Field component	The number in x direction	The number in y direction
E_z	N_x+1	N_y+1
H_x	N_x+1	N_y
H_y	N_x+1	N_y

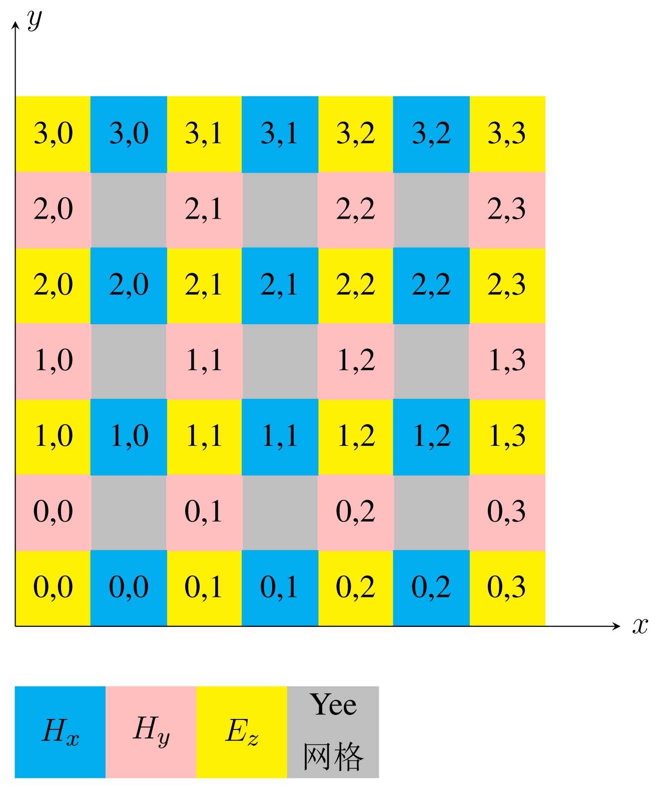


图 3-2 The position of discrete field points in Yee cells of TM mode wave

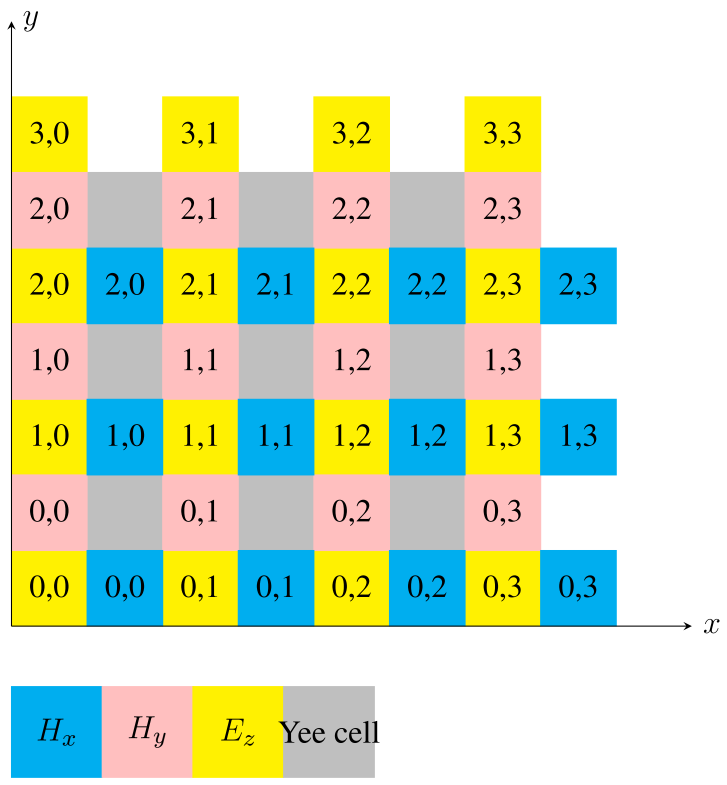


图 3-3 The computational model of traditional data parallelism method

In the program, we allocate a block of memory whose addresses are continuous and size is corresponding the number of the field component whose data is stored in it. For example, in the 3×3 Yee cell, we allocate three blocks of memory which can store 16 floats, 12 floats, 12floats, to store the data of E_z, H_x, and H_y respectively. Besides, we follow the rule that all field vector components are growing in the x direction first, ie the second number of the serial number growing first. So, take discrete field points of E_z as an example, they are stored in the memory in a one-dimension way, and their serial numbers are $(0,0)\cdots(0,3),(1,0)\cdots$.

The process of computing field vector components, here, we explicate it by taking computing H_x as an example, without specify the instruction set used. The first step, load three vectors according to the equation (3-1). The first vector, including the first five discrete points of H_x, ie $H_x(0,0)$ to $H_x(1,0)$, is noted as V_{H_x}. The second vector, including the first five discrete points of E_z, ie $E_z(0,0)$ to $E_z(1,0)$, is noted as $V^1_{E_z}$. The third vector, including five continuous discrete points of E_z from the second points, ie $E_z(0,1)$ to $E_z(1,1)$, is noted as $V^2_{E_z}$.

The second step, we use instructions to have $V^1_{E_z}$ minus $V^2_{E_z}$, and note the result as V_{sub}. More specifically speaking, let the first data of the $V^1_{E_z}$, minus the first data of $V^2_{E_z}$, and so on. Then, all data of V_{sub} multiply the constant, $\frac{\Delta t}{\mu\Delta x}$, and the result is noted as $V^{'}_{sub}$. At last, let $V^{'}_{sub}$ add V_{H_x}, to get the final consequence. So, we implement equation (3-1). Repeating this step over all discrete points of H_x. The process is illustrated in Fig. 3-4.

The last step, is to store those elements of vectors back to memory, a reverse to the first step.

From the Fig. 3-4, we can see that one of those auxiliary points, ie $H_x(0,3)$, keep all elements in those three vector having a constant relationship, which allows us process those elements by one instruction at one time. In Fig. 3-4, the constant relationship is the serial numbers of one set of points of E_z are the same as the serial number of the set of points H_x, while the serial numbers of another set of points of E_z, are bigger by 1 on two parts of serial numbers. If we remove those auxiliary points, we need to add some code to find out which field points we should load in VP, and compute less than 4 points by using scalar processor in each row. Those extra process will make program more complex and fewer efficient.

$$\begin{array}{lll} V^1_{E_z} & V^2_{E_z} & \\ E_z(0,0) \; - & E_z(0,1) \to & H_x(0,0) \\ E_z(0,1) \; - & E_z(0,2) \to & H_x(0,1) \\ E_z(0,2) \; - & E_z(0,3) \to & H_x(0,2) \\ E_z(0,3) \; - & E_z(1,0) \to & H_x(0,3) \\ E_z(1,0) \; - & E_z(1,1) \to & H_x(1,0) \end{array}$$

图 3-4 The process of computing H_x in traditional scheme

3.3 A New Computational Model for Data Parallelism

In the traditional parallel FDTD method, the computational model derived directly from Yee cell showed in Fig. 3 is adaptable to almost all boundary conditions. Nevertheless, for some specific boundary conditions, like PEC and MUR, we can modify the model of the traditional data parallel method.

Let us take a look on PEC boundary condition first. PEC boundary condition is a TBC. In PEC, we set all electric field points are 0. ie

$$E(k) = 0, \tag{3-4}$$

in which $E(k)$ represent the discrete electric field points on boundaries.

The Mur ABC, "absorbing" waves without any reflection when they arrive at the boundary of simulation area's boundaries. Consider the homogeneous equation in one-dimension form:

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0. \tag{3-5}$$

The solution to equation (3.3) is

$$u(x,t) = A\exp[j(\omega t - k_x x)]. \tag{3-6}$$

Set $x = 0$ is the left boundary. For there are incident wave and reflected wave, so we have

$$u(x,t) = A_- \exp[j(\omega t + k_x x)] + A_+ \exp[j(\omega t - k_x x)]. \tag{3-7}$$

Note the first and second term of the right side is u_- and u_+ respectively. If we want to remove the reflected wave, we should set u_+ to be 0. Hence, there has

$$\frac{\partial u}{\partial x} - \frac{1}{c}\frac{\partial u}{\partial t} = 0. \tag{3-8}$$

By discretizing equation (3.3), we have

$$u^{n+1}(k) = u^n(k-1) + \frac{c\Delta t - \Delta z}{c\Delta t + \Delta z}[u^{n+1}(k-1) - u^n(k)], \tag{3-9}$$

In which $u(k)$ represent boundary points, $u(k-1)$ represent the points close to the $u(k)$.

According to equation (3-4) and (3-9), discrete points of other field vector component are not need to compute those points of electric field and on boundaries. Therefore we can simplify the computational model, it the spatial organization shown in Fig. 2-1 by remove some magnetic field points between those electric field points which on boundary. By doing this, the number of discrete points need to be computed is reduced. Here, we still use the 3×3 Yee cell grid to explain our modified computational model, which shown in Fig. 3-5.

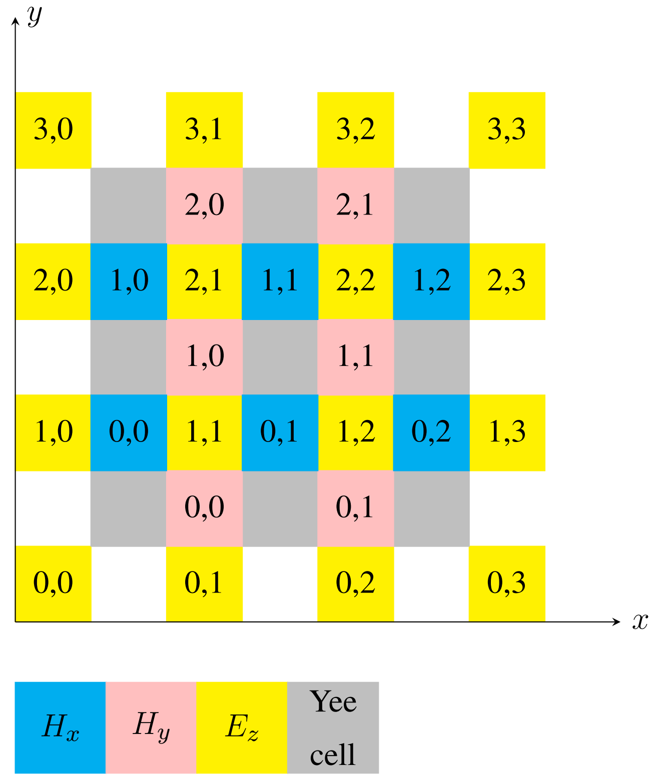


图 3-5 The modified computational model of data parallelism

From the Fig. 3-5 we can see that all magnetic field points outside the Yee cell grids area are removed. Now, the number of discrete field points of each field vector component are shown as Table 3-2. Also let N_x and N_y are the number of Yee cell of x, y direction respectively.

表 3-2 The number of discrete points of each field component in each direction in new computational model

Field component	The number in x direction	The number in y direction
E_z	N_x+1	N_y+1
H_x	N_x	N_y-1
H_y	N_x-1	N_y

In the program, the same with what we doing in traditional scheme, we allocate a block of memory whose addresses are continuous and size is corresponding the number of the field component whose data is stored in it. For example, in the 3×3 Yee cell, we allocate three blocks of memory which can store 16 floats, 6 floats, 6 floats, to store the data of E_z, H_x, and H_y respectively. Besides, we also follow the rule that all field vector components are growing in the x direction first, ie the second number of the serial number growing first.

In the traditional computational model, with those auxiliary H_x points, the physical position of a data in memory is uniform to its logical position, ie the m-th float stored in three blocks memory are $E_z(y_m,x_m)$, $H_x(y_m,x_m)$, and $H_y(y_m,x_m)$. However, in the modified new computational model things get changed. So, taking the computation of H_x as example again. According the steps stated in the last section, when we want to compute the first four points of H_x, we need to load three vectors in VP. They are V_{H_x}, including $H_x(0,0)$ to $H_x(1,0)$; $V^1_{E_z}$, including $E_z(1,0)$ to $E_z(1,3)$; and $V^2_{E_z}$, including $E_z(1,1)$ to $E_z(2,0)$. Following the process mentioned in the last section, let the $V^1_{E_z}$ minus $V^2_{E_z}$. You will find there are something wrong here: The $E_z(2ﬁ0)$ will minus $E_z(1,3)$. However, according to the equation (3-1), we need $E_z(2,1)$ to minus $E_z(2,0)$ if we want to update the value of $H_x(1,0)$.

As things are becoming different now, we have to do two modifications. The first, we can no more apply one process over all points of a field vector component. We should divide a rwo into several separate sections and do different processes on them separately, too. The second, we should understand that not all points can be computed by VP, now.

We should note that some special points, like some point on head or tail of a row.

3.4 The Comparison between Traditional and Modified Computational Model

In this paper, we take the computation of 2D FDTD TM wave using Mur ABC as an example to analyze the difference between using data parallelism or not, the traditional and our modified method, and the performance of the modified method in different space and time sizes. Furthermore, we analyze the efficiency of those two methods based on those results.

The framework of the program is presented in the function’s call tree form, which is illustrated in Fig. 3-6. From it, we can see that the program consists of 3 parts which are input, response to initialize data; `H\_cmp` and `E\_cmp`, two functions that compute electromagnetic field; and other functions that dealing with other things. Once we run the code, the Input works first and then are `H\_cmp` and `E\_cmp` which are called iterating, the last,other functions do some auxiliary works like saving the result of the computation to files.

We adopted the methods provided by Visual Studio Profiling Tools to collect the performance data of sample project and analyze them. There are two kinds of profiling methods we were using. The first is sampling profiling method, and another is instrumentation profiling method. The sampling profiling method collects statistical data about an application, including the function call stack. The instrumentation profiling method collects detailed timing for the function calls in a profiled application.

Instrumentation use four types of values to represent the time spent in a function or source code line and here we use two of them, which are:

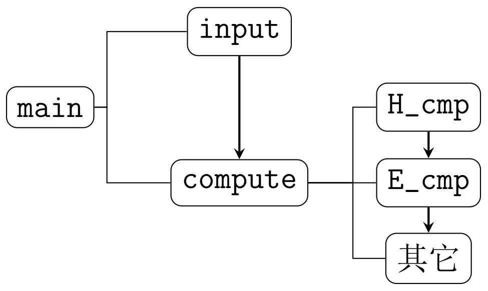


图 3-6 The framework of the program

Elapsed Inclusive The total time that is spent to execute the function or source line.

Elapsed Exclusive The time that is spent executing code in the body of the function or source code line. Time that is spent executing functions that are called by the function or source line is excluded.

So, in our program, the elapsed inclusive time of function `main` represent the time elapsed by the whole program; the elapsed inclusive time of function epresent the time elapsed to compute all discrete field points; as `H_cmp` and `E_cmp` do not call other functions, so the elapsed inclusive and exclusive time are equal and all represent the average time spent on those two functions. In fact, for function `H_cmp` and `E_cmp`, we concern exactly the time spent in one time.

We also use Release option to see the differences more clearly, as by doing this we can ignore some minor problems caused by accident or bad programming habit. Besides, to guarantee the accuracy of the analysis, for every condition we ran the program for five times and obtain the average value of those profiling results. The configuration is Intel Core i5-3210M 2.3GHz, 4.00GB memory, Windows 10 64-bits and Visual Studio 2015.

The following tables give out the performances of traditional scheme and our modified scheme which using new computational model under different condition.

表 3-3 The comparison between using data parallelism or not

Function	Elapsed Exclusive (ms)		Time saved by using data parallelism
	No parallelism	New model	
`main`	10330.43	7835.78	24.15%
`compute`	10313.62	7819.76	24.18%
`H_cmp`	6.16	4.64	24.62%
`E_cmp`	4.10	3.13	23.78%

[1] The size of simulation area is 2000×1000 Yee cells.

[2] The number of time step is 1000.

According to Table 3-3. The whole running time of `H_cmp` and `E_cmp` are obtained by multiplying the time steps with the Average Elapsed Exclusive Time. We can see easily after some simple calculations that the time spent by those two functions are 6160ms and 4100ms, over 99.3% and 99.2% of the time spent by the whole program respectively.

Hence, we can say safely that the differences between two methods are mainly caused by the `H_cmp` and `E_cmp`. According to the Table 2, the time-consuming of FDTD method had been reduced about by 24%. It shows us that data parallelism can enhance the efficiency of FDTD largely.

表 3-4 The comparison between traditional and new computational model

Function	Elapsed Inclusive (ms)		Time saved by using new model
	Traditional model	New model	
`main`	4229.46	4082.64	3.47%
`compute`	4216.81	4070.73	3.46%
`H_cmp`	2.50	2.41	3.37%
`E_cmp`	1.68	1.62	3.57%

[1] The size of simulation area is 1000×1000 Yee cells.

[2] The number of time step is 1000.

What we can see from the Table 3-4 is that the elapsed time of the whole program by using new computational had been saved about by 3.45% compared to the traditional model. With the analysis we did above, the conclusion can be concluded that the computational model of the new methods is better than that of the traditional method. Though some discrete field points are computed by scalar processor, which is a drawback, but the reduction of the number of whole discrete points which means less points the processor need to compute, not only compensate the waste caused by that drawback, but also make the elapsed time shorter.

表 3-5 The performance of new model under different space size

Function	Elapsed Inclusive (ms)		
	1000×1000	2000×1000	3000×1000
`main`	4082.64	7835.78	11597.80
`compute`	4072.73	7819.76	11577.88
`H_cmp`	2.412	4.64	6.87
`E_cmp`	1.62	3.13	4.65

[1] The number of time step is 1000.

Based on Table 3-5, we can see the time-consuming of the whole program is linear to the size of space scale. From the data we collected, when the size of space scale is as n times as before, the time-consuming will be about $0.92n + 0.08$ times than before.

表 3-6 The performance of new model under different time size

Function	Elapsed Inclusive (ms)	
	1000 time steps	3000 time steps
`main`	7835.78	23247.73
`compute`	7819.76	23230.83
`H_cmp`	4.64	4.59
`E_cmp`	3.13	3.10

[1] The simulation area had 2000×1000 Yee cells.

From the Table 3-6, we can see that with time step increases, the average time-consuming of every time step is decreased. The time-consuming of each time step in the 3000 steps condition is 0.99 times to that of 1000 time steps condition. We believe that this is because with the number of iteration increase, the computer can hit the memory more accurately.

3.5 Conclusion

In this chapter, the 2D TM model is taken as an instance to explain a new data parallel FDTD method which is a modification of the traditional methods and how it is advanced under MUR boundary condition with fewer boundary points that need to be compute. The new method was programmed by using C and SSE. Besides, based on the profiling results by running the program for different conditions like the different sizes of simulation space and time, the performance data of the new method and the traditional method were obtained. At last, by analyzing the performance data, the conclusion was derived that our new method saves about 3.45% time which mean that it is better than the traditional data parallel FDTD method.

Chapter 4 Accelerating FDTD by Using CUDA

In this chapter, we use the simulation of 2D TM mode as an instance to explain the scheme of accelerating FDTD by using CUDA.

4.1 GPU and CUDA

GPU, is a specialized microprocessor designed to process images rapidly with hundreds of cores and independent memory. The term GPU was popularized by NVIDIA corporation in 1999. GPU was spread with the population of graphic operation systems.

In the early time, for RAM was very expensive, the graphic chips composited data together. Till 80s, the Commondore Amiga featured a custom graphic chip which can manipulate 16-color bitmaps. In 1986, the TMS34010 was released by Texas Instruments, which is the first microprocessor could run general purpose code by using a very graphics-oriented instruction set. OpenGL is a professional graphic application programming interfae (API) appeared in the early 90s, which tried to be a standard cross-language, cross-platform API to rendering 2D and 3D vector graphics. In 2001, the NVIDIA developed the first consumer-level GPU, GeForce 256, which is programmable and supporting OpenGL and DirectX8. In 2006, NVIDIA released CUDA. Against previous API like Direct3D and OpenGL which required advanced skills in graphics programming and advanced understanding of the hardware' architecture, it is designed to work with programming language such as C, C++, and Fortran, which allow people utilize GPU's computation power easier without requiring so much.

In 2006, NVIDIA released GeForce8800 GTX, which is a milestone of the development of general purpose GPU (GPGPU). From that time, GPU began to have the architecture which suitable to execute general purpose computations, and including several new components like shared memory, multithreading communications which give it the powerful capability of parallel computing.

In 2007, CUDA was unveiled by NVIDIA. Unlike precious generations that partitioned computing resources into vertex and pixel shaders, the CUDA included a unified shader pipeline, which allowing wach and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations. Besides, those ALUs were built to comply with IEEE requirements for single-precision floating-point arithmetic and were designed to use an instruction set tailored for general computation rather than specifically for graphics. Furthermore, the execution units on CUDA were allowed arbitrary read and write access to memory as well as access to a sofeware-managed cache known as shared memory. To make the using of GPU more easier, NVIDIA took standard C added a small number of keywords to use GPU. Hence, users are no longer required to have any knowledge of the OpenGL or Direct3D, nor the knowledge to transform their tasks to look like a computer graphics task in order to be executed by GPU.

4.2 The Theory of Accelerating FDTD with CUDA

4.2.1 The Programming Model of CUDA

As GPU is an independent device, so the system of GPU is independent to the system of CPU. Generally, we refer to the CPU and the system's memory as the *host*, and refer to the GPU and its memory as the *device*. The code on host mainly control the work flow of a program, and the code on device often focus on the tasks involving massive computation.

CUDA allows the programmer to define C functions called *kernels*, which when called, are executed N times in parallel by N different CUDA threads, against only once like regular C functions. When a CUDA program is being executed, kernels are called to use GPU to compute, that is, at first transferring data to GPU, and transferring the result of computing those data back after the tasks are completed. The process is illustrated in Fig. 4-1.

In the following, we will introduce the CUDA programming model by a demonstration to show how to accelerating FDTD with CUDA. In the following code, we write a function which intend to add two vectors.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
```

```
#include <stdio.h>
#define N (33 * 1024)
__global__ void add(int *a, const int *b, const int *c)
{
    int tid=threadIdx.x + blockId.x * blockDim.x;
    while(tid < N){
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main()
{
    //declare for both Host and Device
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    //allocate memory for Device
    cudaMalloc(&dev_a, N * sizeof(int));
    cudaMalloc(&dev_b, N * sizeof(int));
    cudaMalloc(&dev_c, N * sizeof(int));

    //Initialize Host data
    for (int i = 0; i < N; i++){
        a[i] = i;
        b[i] = i * i;
    }

    //Transfer Host data to Device.
    cudaMemcpy(dev_a, a, N*sizeof(int),
       cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int),
       cudaMemcpyHostToDevice);

    //Excute Kernel
    add <<<128, 128 >>>(dev_a, dev_b, dev_c);

    //Transfer result from Device to Host
    cudaMemcpy(c, dev_c, N*sizeof(int),
       cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

At first, we define a kernel by using the declaration specifier `__global__`, and execution configuration syntax `<<<...>>>`, which specify the number of CUDA threads that execute the kernel. Each thread the executes the kernel has a unique thread ID through the built-in `threadIdx` variable which can be accessible within the kernel. Before calling a kernel, we should allocate memory on device first for three arrays. Correspondingly, we should free the memory allocated when they are no longer needed. Besides, as we mentioned before, the memories of device and host are independent, so we need transfer the data waited to be manipulated to kernel before calling it and transfer the computing result after the kernel being called.

4.2.2 Thread Hierarchy

The basic unit of parallel computing is thread. As mentioned above, thread had its own `threadIdx`, a 3-component vector. So, we can identify thread by using a one-dimensional, two-dimensional, or three-dimensional thread index, to from a one-dimensional, two-dimensional, ot three-dimensional block of threads, called a thread block. The threads in the same thread block can share memory and communicate with each other rapidly. The number of block and threads per block has is defined by arguments in `<<<...>>>`. The first argument in the chevrons specify the number of thread blocks, and the second argument specify the number of threads per thread block has. In the sample code above, the host code in line 36 invoke 128 thread blocks with 128 threads per thread block. The GPU will create a copy of function `add()` for each thread and process them in a parallel way.

In the line 7 of the sample code, there are some built-in variables, which is `threadIdx.x`, `blockId.x`, and `blockDim.x`. For those built-in variables, there is no need to define it or assign a value to it. It contains the value of the index or the size for a thread or block. For a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + yD_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$. In the Fig. 4-2, we illustrate the relationship between a two-dimensional block with threads it has.

After all threads finished their copy of kernel, we add the theirs index with the number of all threads in all blocks, like what line 10 doing in the sample code. By doing this, we can compute vectors in arbitrary size with finite number of threads and blocks.

4.3 The Scheme of Accelerating FDTD with CUDA

Before we consider the scheme of kernels, we think some optimizations for auxiliary work.

First, we consider the transfer step. As mentioned above, before calling a kernel, we need transfer the data which need to be manipulated to device, and transfer the result back to host. This process will bring us some inconveniences.

The transfer step is accomplished in a serial way, which is inefficient, especially considering that almost all kernel need to handle massive data and the limitation of the I/O capability of hardwares, thus the CPU and GPU will have a long time with nothing meaningful to do, wasting their power of computation.

For this problem, we adopt a measure that instead of transfer data from host to device, we initialize data on device directly after allocating memory on it. By doing this, half time of transferring data is saved.

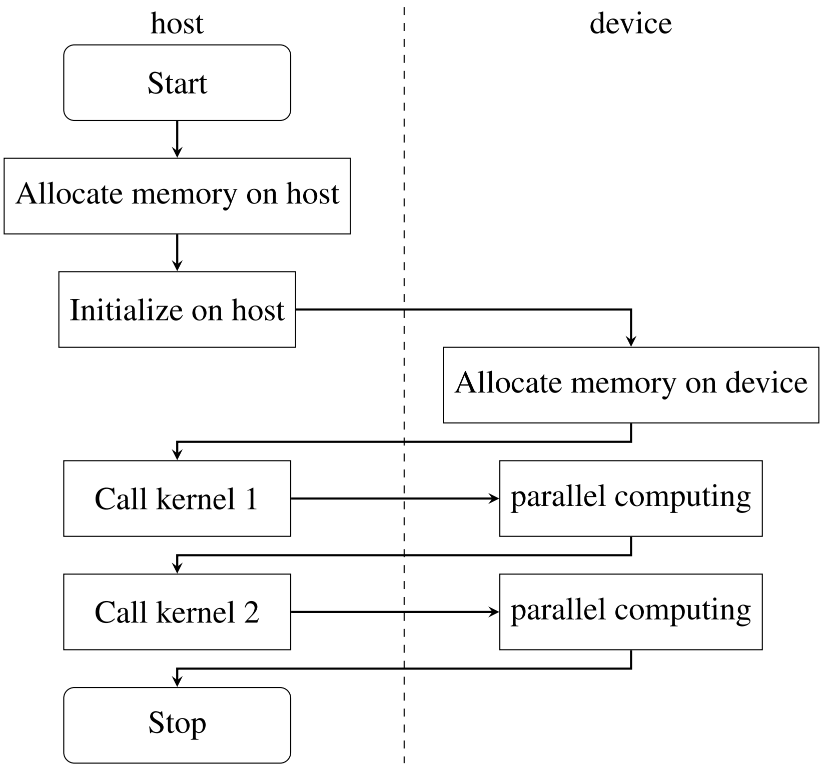


图 4-1 The process of a CUDA program

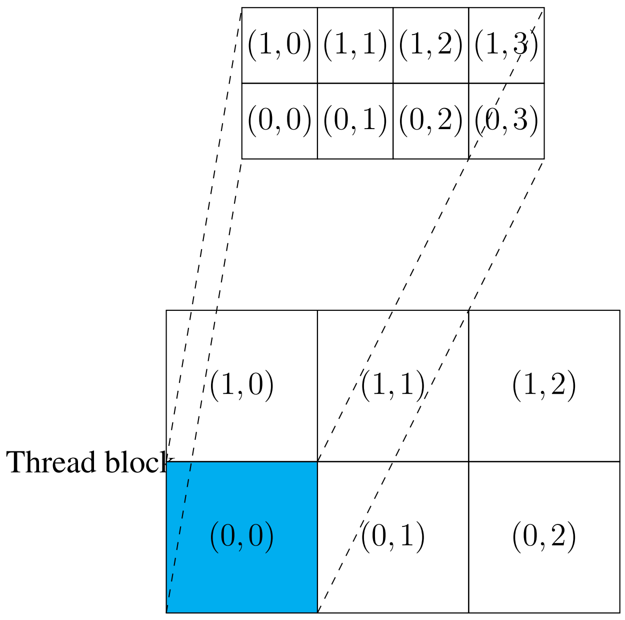


图 4-2 The relation between a two-dimensional thread block and threads it has

Second, we consider the memory of device. Considering we need to have a 2D arrays' allocations, we performing pitch allocations using `cudaMallocPitch` instead of `cudaMalloc`. The alignment in memory had been mentioned in the chapter 2 that if the allocations are not aligned much time would be wasted on accessing memory and loading data. The `cudaMallocPitch` ensure that the starting address of each row in a 2D array is a multiple of 2^N, where N is $7\tilde{1}0$. In the worst condition, the time can be 1.5 times of not aligned condition compared to the aligned condition. For the same reason, we use `cudaMemcpy2D` instead of `cudaMemcpy` to get the result back from device to host.

Together with the optimizations mentioned above, we draw the framework of FDTD program, which shown in Fig. 4-3. There are two parts of the CUDA program, one part is classes, and another is functions. In the classed part, three classed are defined. The members of class `E` include the information of electric field, and some method like initialization, checking, and saving the data of electric field in files. The class `H`, like class, define the same thing to the H_x and H_y field components. The class `src`, on the other hand, specify the information and methods of the whole simulation area, including the excitation source, the scale of Yee cells, the number of time steps, etc.

The function part, also the key part of the program, is responsible to the process of updating values of each discrete field point, including the updating of magnetic field points, the updating of electric field point, and the process of boundary condition in each tiem step.

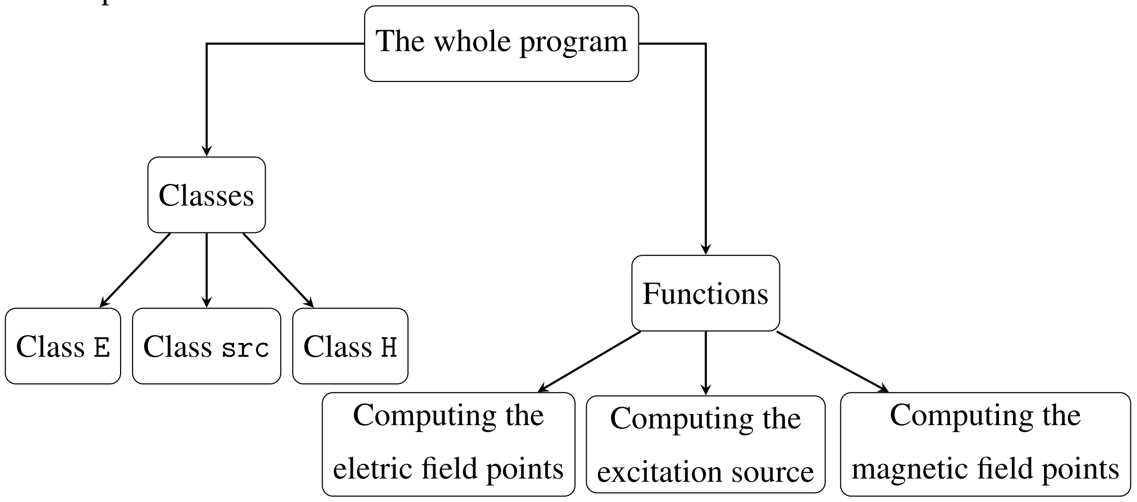

The whole program
Classes
Functions
Class E
Class src
Class H
Computing the eletric field points
Computing the excitation source
Computing the magnetic field points

图 4-3 The framework of FDTD program with using CUDA

The work flow of the program if shown in Fig. 4-4. At the beginning we initialize the simulation area, plus the class `E` and `E`. Then we step into the loop of updating values of each discrete field point. After the loop the program get the result back from device and write the information of field components in files. At last, free the allocations of memory and end the program.

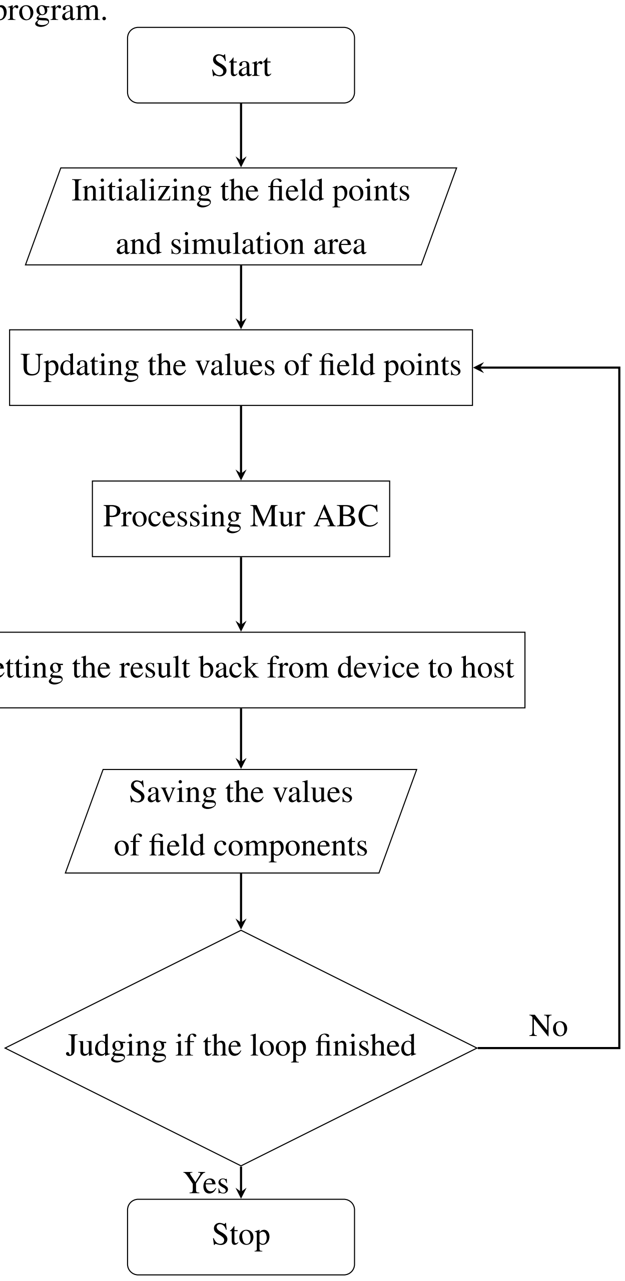

Start
Initializing the field points and simulation area
Updating the values of field points
Processing Mur ABC
Getting the result back from device to host
Saving the values of field components
Judging if the loop finished
No
Yes
Stop

图 4-4 The work flow of program

In the process of updating values, we adopt a way following the programming model discussed above. Taking the code of updating process of E_z shown in the following as an example:

```
__global__
```

```
void Ez_cmp_kernel(
float* Ez, float* Hx, float* Hy,
float coe_Ez, int size_Ez_x, int size_Ez_y,
int ele_ex, int ele_hx, int ele_hy
)
{
    int x, y;
    int tid, number;
    tid = threadIdx.x + blockIdx.x*blockDim.x;
    float dif_Hy, dif_Hx;
    while (tid < ele_ex*size_Ez_y)
    {
        number = tid + 1;
        y = number % ele_ex;//row
        x = number - (y*ele_ex);//column
        //Hy(i,j)   -   Hy(i-1,j)
        dif_Hy = Hy[y*ele_hy + x] - Hy[(y - 1)* ele_hy +
           x];
        //Hx(i,j-1) -   Hx(i,j)
        dif_Hx = Hx[y*ele_hx + (x - 1)] - Hx[y*ele_hx + x
           ];
        Ez[y*ele_ex + x] += coe_Ez * (dif_Hx + dif_Hy);
        tid += blockDim.x*gridDim.x;
    }
}
```

4.4 The performance of FDTD with CUDA

Here, we also take the computation of 2D FDTD TM wave using Mur ABC as an example to analyze the difference of performance between FDTD without data parallelism, the modified method with VP, and FDTD with CUDA in different space and time sizes. Furthermore, we analyze the efficiency of those two methods based on those results.

The call tree view of program is as what Fig. 4-5 showing. We also adopt the profiling methods used in chapter 3. The following tables present the profiling reports.

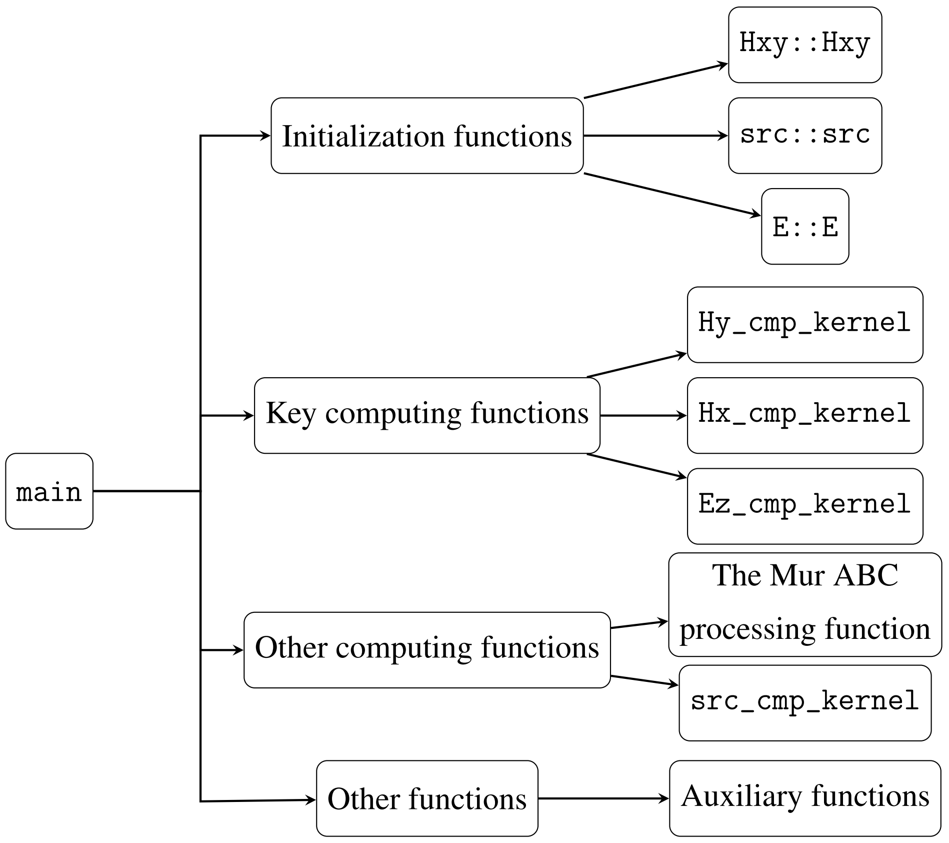


图 4-5 The call tree view of program

The Table 4-1 represent the performances of using serial way and using CUDA. What we can see from it is the time elapsed by computational functions, ie H_cmp and E_cmp takes significant part of the whole time in the serial way, but takes only less than 1% of the whole time in using CUDA way. In fact, in the CUDA accelerating scheme, in each time step, for each discrete field points we use a individual thread to execute it, which means that all values of points are updated at the same time. In theory, if the number of threads in all thread blocks is Sum_{thrd}, the time used in computation by using CUDA is $1/Sum_{thrd}$ times of it in serial way. From the profiling report we also can see that, the time used to update values of filed points takes only 0.81$\%$, and almost all time were taken to allocate memory.

The Table 4-2 represent the comparison between data parallelism and CUDA. From the comparison we can know that the time taken by updating values is still less than 1%.

表 4-1 The comparison between serial way and using CUDA

Function	Elapsed Inclusive (ms)		The time saved by using CUDA
	serial way	using CUDA	
$main$	10330.43	888.30	91.4%
H_cmp[3]	6.16		
E_cmp[3]	4.10		
Hy_cmp_kernel[4]		<0.01	
Hx_cmp_kernel[4]		<0.01	
Ez_cmp_kernel[4]		<0.01	

[1] The size of simulation area is 2000×1000 Yee cells.

[2] The number of time step is 1000.

[3] Only in serial way.

[4] Onely in the way of using CUDA.

表 4-2 The comparison between the modified data parallelism and using CUDA

Function	Elapsed Inclusive (ms)		The time saved by using CUDA
	The modified data parallelism	Using CUDA	
$main$	4082.46	6203.20	-51.9%
H_cmp[3]	2.41		
E_cmp[3]	1.62		
Hy_cmp_kernel[4]		0.02	
Hx_cmp_kernel[4]		0.01	
Ez_cmp_kernel[4]		0.01	

[1] The size of simulation area is 1000×1000 Yee cells.

[2] The number of time step is 1000.

[3] Only in serial way.

[4] Onely in the way of using CUDA.

In Table 4-3 we can see that even the size of simulation area increase, the time used to update values is still very short, even too short to measure it. Compared to the Table 4-2, it is obviously that even accelerate by VP, the execution is still slow in hundreds times compared to using CUDA.

From Table 4-4, we can see that in different number of time iterations, the CUDA scheme can also accomplish the computational task in a very short time. From the profiling report, we can know that the differences in elapsed time between those different conditions are caused by the function `cudaMallocPitch`, which need to allocate memory for arrays. This function, takes about 90% time of the whole time. No more details revealed by NVIDIA corporation so far, so we do not understand the reason behind this situation.

4.5 Conclusion

In this chapter, we introduced the architecture of CUDA, and the programming model used to accelerating FDTD with using CUDA. Then we explicate our CUDA scheme, which taking the computation of 2D FDTD TM wave using Mur ABC as an example, and profiling the performance of our scheme. Furthermore, we analyze the profiling reports together the reports given in previous chapters. In the analysis, we found that the potential of CUDA is substantial. However, we also noticed that as an independent device, we need to design the CUDA code more prudentially.

表 4-3 The performance of CUDA under different size of simulation area

Function	Elapsed Inclusive (ms)		
	1000×1000	2000×1000	3000×1000
$main$	6203.20	883.30	923.67
Hy_cmp_kernel	0.02	<0.01	<0.01
Hx_cmp_kernel	0.01	<0.01	<0.01
Ez_cmp_kernel	0.01	<0.01	<0.01

[1] The number of time step is 1000.

表 4-4 The performance of CUDA under different number of time steps

Function	Elapsed Inclusive (ms)	
	1000 time steps	3000 time steps
$main$	833.30	1139.94
$H_y\_cmp\_kernel$	<0.01	<0.01
$H_x\_cmp\_kernel$	<0.01	<0.01
Ez_cmp_kernel	<0.01	<0.01

[1] The size of simulation area is 2000×1000 Yee cells.

Chapter 5 Conclusion

In this paper, we focus on the topic of accelerating FDTD with hardwares. First, we discussed accelerating FDTD with a built-in CPU component, VP. For using VP, the theory and a practical scheme were discussed in details. Furthermore, we modified the traditional data parallelism scheme by removing some unnecessary discrete field points. To test the advantage of our modified scheme, the 2D FDTD in TM mode were taken as an example. After the experience, we analyzed the profiling reports. Then, for executing FDTD with CUDA, we do the same things. After that, the conclusion we obtained is that the modified data parallelism scheme is better than traditional scheme, however, compared to the CUDA, it is still very inefficient. The CUDA is a powerful potential computational power waiting to be utilized in the future.

Nevertheless, there are still some places waiting to be enhanced. For example, using constant memory to store some constants, dividing the simulation area into some sub-areas.

In the research of this topic, I have learned much knowledge about FDTD algorithm, CUDA, and vector processor. Gained some valuable experience of dealing with a project has complex organization and huge size. Besides, learned the rules of doing research, including how to collect previous research contributions, find the point to enhance, examine new ideas etc., which is the solid foundation for future growth.

Acknowledgements

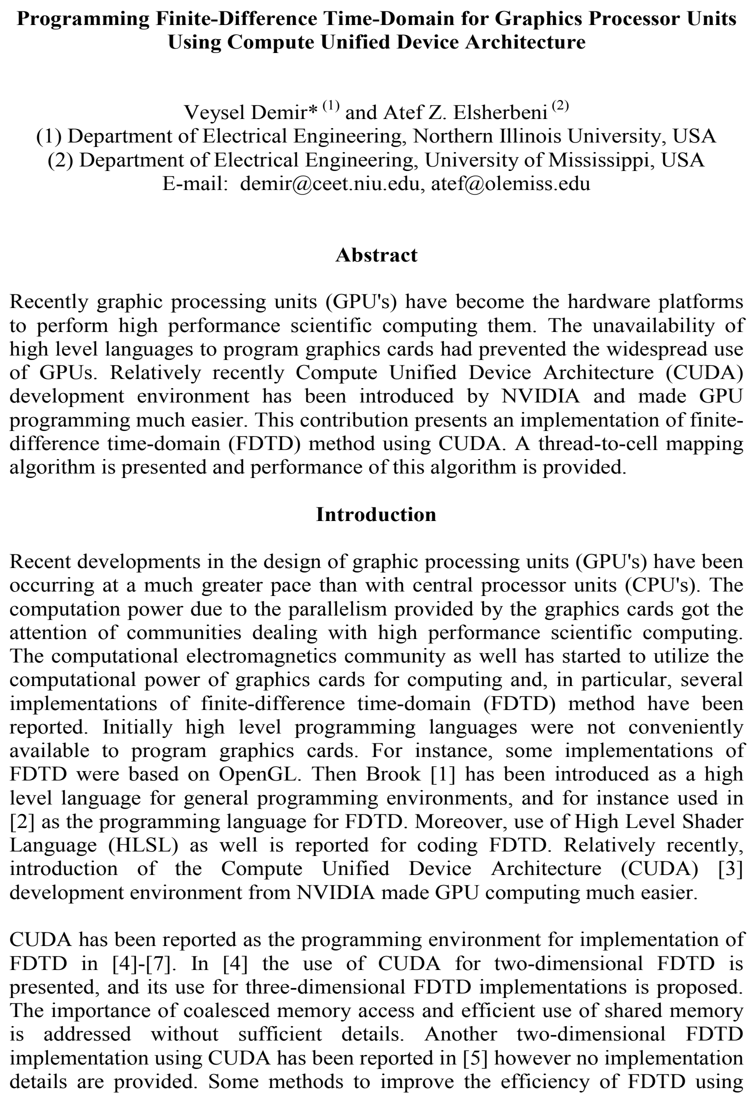
在攻读博士学位期间，首先衷心感谢我的导师 XXX 教授，……
……

外文资料原文

1.1 外文资料信息

V. Demir and A. Z. Elsherbeni, ”Programming finite-difference time-domain for graphics processor units using compute unified device architecture,” 2010 IEEE Antennas and Propagation Society International Symposium, Toronto, ON, 2010, pp. 1-4. doi: 10.1109/APS.2010.5562117

1.2 外文资料原文



Programming Finite-Difference Time-Domain for Graphics Processor Units Using Compute Unified Device Architecture

Veysel Demir* [(1)] and Atef Z. Elsherbeni [(2)]
(1) Department of Electrical Engineering, Northern Illinois University, USA
(2) Department of Electrical Engineering, University of Mississippi, USA
E-mail: demir@ceet.niu.edu, atef@olemiss.edu

Abstract

Recently graphic processing units (GPU's) have become the hardware platforms to perform high performance scientific computing them. The unavailability of high level languages to program graphics cards had prevented the widespread use of GPUs. Relatively recently Compute Unified Device Architecture (CUDA) development environment has been introduced by NVIDIA and made GPU programming much easier. This contribution presents an implementation of finite-difference time-domain (FDTD) method using CUDA. A thread-to-cell mapping algorithm is presented and performance of this algorithm is provided.

Introduction

Recent developments in the design of graphic processing units (GPU's) have been occurring at a much greater pace than with central processor units (CPU's). The computation power due to the parallelism provided by the graphics cards got the attention of communities dealing with high performance scientific computing. The computational electromagnetics community as well has started to utilize the computational power of graphics cards for computing and, in particular, several implementations of finite-difference time-domain (FDTD) method have been reported. Initially high level programming languages were not conveniently available to program graphics cards. For instance, some implementations of FDTD were based on OpenGL. Then Brook [1] has been introduced as a high level language for general programming environments, and for instance used in [2] as the programming language for FDTD. Moreover, use of High Level Shader Language (HLSL) as well is reported for coding FDTD. Relatively recently, introduction of the Compute Unified Device Architecture (CUDA) [3] development environment from NVIDIA made GPU computing much easier.

CUDA has been reported as the programming environment for implementation of FDTD in [4]-[7]. In [4] the use of CUDA for two-dimensional FDTD is presented, and its use for three-dimensional FDTD implementations is proposed. The importance of coalesced memory access and efficient use of shared memory is addressed without sufficient details. Another two-dimensional FDTD implementation using CUDA has been reported in [5] however no implementation details are provided. Some methods to improve the efficiency of FDTD using

图 A-1 外文原文资料第一页

CUDA are presented in [6], which can be used as guidelines while programming FDTD using CUDA. The discussions are based on FDTD updating equations in its simplest form: updating equations consider only dielectric objects in the computation domain, the cell sizes are equal in *x*, *y*, and *z* directions, thus the updating equations include a single updating coefficient. The efficient use of shared memory is discussed, however the presented methods limits the number of threads per thread block to a fixed size. The coalesced memory access, which is a necessary condition for efficiency on CUDA, is inherently satisfied with the given examples; however its importance has never been mentioned.

In this current contribution a CUDA implementation of FDTD is provided. The FDTD updating equations assume more general material media and different cell sizes. A thread-to-cell mapping algorithm is presented and its performance is provided.

FDTD Using CUDA

The unified FDTD formulation [8] considered for CUDA. The problem space size is $Nx \times Ny \times Nz$, where Nx, Ny, and Nz are number of cells in *x*, *y*, and *z* directions, respectively. Thus, for instance, the updating equation that updates *x* component of magnetic field is given in [8] as

$$\begin{aligned} H_x^{n+\frac{1}{2}}(i,j,k) &= C_{hxh}(i,j,k)H_x^{n-\frac{1}{2}}(i,j,k) \\ &+C_{hxey}(i,j,k)\left(E_y^n(i,j,k+1)-E_y^n(i,j,k)\right). \\ &+C_{hxez}(i,j,k)\left(E_z^n(i,j+1,k)-E_z^n(i,j,k)\right) \end{aligned} \tag{1}$$

In order to achieve parallelism, the threads are mapped to cells to update them. For the mapping, a thread block is constructed as a one-dimensional array, as shown on the first two lines in Listing 1, and the threads in this array are mapped to cells in an *x-y* plane cut of the FDTD domain as illustrated in Fig. 1. In the kernel function, each thread is mapped to a cell; *thread index* is mapped to *i* and *j*. Then, each thread traverses in the *z* direction in a *for* loop by incrementing *k* index of the cells. Field values are updated for each *k*, thus the entire FDTD domain is covered.

```
block_dim_x = number_of_threads; block_dim_y = 1;
n_blocks_y = 1;
n_blocks_x = (nxx*nyy)/number_of_threads +
                              ((nxx*nyy)%number_of_threads ==0?0:1);
```

Listing 1. CUDA code to define block and grid sizes.

Unfortunately in FDTD updates the operations are dominated by memory accesses. In order to ensure high performance all global memory accesses shall be coalesced. In general an FDTD domain size would be an arbitrary number. In order to achieve coalesced memory access, the FDTD domain is extended by padded cells such that the number of cells in *x* and *y* directions is an integer

图 A-2 外文原文资料第二页

multiple of 16 as illustrated in Fig 2. The modified size of the FDTD domain becomes $Nxx \times Nyy \times Nz$, where Nxx, Nyy, and Nz are number of cells in *x*, *y*, and *z* directions, respectively.

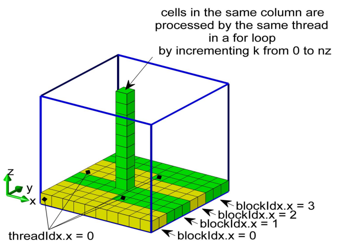


Figure 1. Mapping of threads to cells of an FDTD domain.

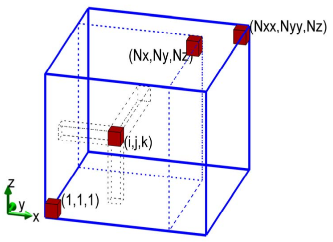


Figure 2. An FDTD problem space padded with additional cells.

```
__global__ void update_magnetic_fields_on_kernel(...)
{
extern __shared__ float sEz[];
int ci = blockIdx.x * blockDim.x + threadIdx.x;
int j  = ci/nxx;
int i  = ci - j*nxx;
int si = threadIdx.x;
int cizp;
float ex, exzp, ey, eyzp;
ey = Ey[ci];
ex = Ex[ci];
for (int k=0;k<nz;k++)
{
       cizp   = ci + nxx*nyy;
       exzp   = Ex[cizp];
       eyzp   = Ey[cizp];
       sEz[si]= Ez[ci];
       if (threadIdx.x<16)
              Ez[blockDim.x+threadIdx.x] = Ez[ci+blockDim.x];
       __syncthreads();

       Hx[ci] = Chxh[ci]*Hx[ci]+ Chxey[ci]*(eyzp-ey)
                          + Chxez[ci]*(Ez[ci+nxx]-sEz[si]);
       Hy[ci] = Chyh[ci]*Hy[ci]+ Chyez[ci] * (sEz[si+1]-sEz[si])
                          + Chyex[ci] * (exzp-ex);
       ...
       ci = cizp;
       ey = eyzp;
       ex = exzp;
       }
}
```
Listing 2. A section of CUDA code to update magnetic field components.

After ensuring the coalesced memory access, data reuse in a *for* loop in *z* direction and appropriate use of shared memory a CUDA code is developed. A section of this code is shown in Listing 2. The developed algorithm is tested on an

图 A-3 外文原文资料第三页

NVIDIA® Tesla™ C1060 Computing card. Size of a cubic FDTD problem domain has been swept and the number of million cells per second processed is calculated as a measure of the performance of the CUDA program. The result of the analysis is shown in Fig. 3. It can be observed that the code processes about 450 million cells per second on the average.

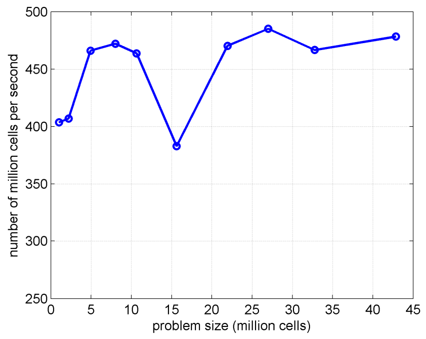


Figure 3. FDTD algorithm speed versus problem size.

References

[1] Buck, *Brook Spec v0.2* Stanford, CT: Stanford Univ. Press, 2003.
[2] M. J. Inman and A. Z. Elsherbeni, "Programming Video Cards for Computational Electromagnetics Applications," *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, pp. 71–78, December 2005.
[3] NVIDIA CUDA ZONE, http://www.nvidia.com/object/cuda_home.html.
[4] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "High-speed FDTD Simulation Algorithm for GPU with Compute Unified Device Architecture," *IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*, *2009*, North Charleston, SC, United States, p. 4, 2009.
[5] Valcarce, G. De La Roche, A. Jüttner, D. López-Pérez, and J. Zhang,"Applying FDTD to the coverage prediction of WiMAX femtocells," *EURASIP Journal on Wireless Communications and Networking*, February 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] Ong, M. Weldon, D. Cyca, and M. Okoniewski, "Acceleration of Large-Scale FDTD Simulations on High Performance GPU Clusters," *2009 IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*, North Charleston, SC, United states, 2009.
[8] Atef Elsherbeni and Veysel Demir, "The Finite Difference Time Domain Method for Electromagnetics: With MATLAB Simulations," SciTech Publishing, 2009.

图 A-4 外文原文资料第四页

外文资料翻译

1.1 外文资料信息

V. Demir 和 A. Z. Elsherbeni，《在使用统一计算架构的图形处理器中进行时域有限差分编程》， 2010 IEEE Antennas and Propagation Society International Symposium, Toronto, ON, 2010, pp. 1-4. doi: 10.1109/APS.2010.5562117

1.2 外文资料正文翻译

1.2.1 摘要

最近图形处理器（GPU）称为执行高性能科学计算的平台。高级程序语言不能对显卡编程曾经阻碍了 GPU 的普及。最近由 NVIDIA 公司引入了统一计算架构（Compute Unified Device Architecture，CUDA）开发环境，使得 GPU 编程容易了许多。本文的工作展示了使用 CUDA 的时域有限差分（Finite-Difference Time-Domain，FDTD）的实现，展现了一个线程到元胞的映射算法和该算法的性能。

1.2.2 引言

近来相比于 CPU，GPU 设计的发展速度相比很快。由于提供并行而带来的计算能力引起了大众对于使用 GPU 进行科学计算的关注。计算电磁学的研究者们也开始使用显卡的计算能力来进行计算，尤其是对 FDTD 的计算已经有研究报告。起初高级语言在显卡编程上很不方便。比如一些 FDTD 的实现是基于 OpenGL 的。之后有 Brook [1] 作为一个面向通用编程环境的高级语言被提出并且在 [2] 中作为 FDTD 的编程语言。更进一步，使用高级着色器语言（High Level Shader Language，HLSL）编写 FDTD 也有报导。最近由 NVIDIA 提出的 CUDA 开发环境让 GPU 编程大幅简化。

CUDA 在 [4]-[7] 中被提出作为实现 FDTD 的编程环境。在 [4] 中，展示了使用 CUDA 的二维 FDTD，并提出了三维的 FDTD 实现方案。强调了使用级联内存的重要性和使用共享内存的高效性但没有进一步详细说明。另一个使用 CUDA 的

二维 FDTD 实现方案在 [5] 中被提出，不过没有提供任何细节。一些使用 CUDA 来提升 FDTD 的效率的方法在 [6] 中提出，可以作为使用 CUDA 对 FDTD 编程的指导大纲。讨论是基于 FDTD 迭代方程的最简形式：仅考虑在计算域中的电解质导体的迭代方程，在各个x，y和z方向上元胞尺寸相同，因此迭代方程只包含一个迭代系数。共享内存的高校也被讨论了，但是展示的方法中每个线程块中的线程数目是有固定大小的。级联内存是 CUDA 在效率上的必须，在给出的例子中内含的满足了，但是重要性未被提及。

本文提出了一个 FDTD 实现方案。其中 FDTD 迭代方程假设了更一般的介质和不同的元胞尺寸。提出了一个线程到元胞的映射算法以及其性能。

1.2.3 使用 CUDA 的 FDTD

统一的 CUDA 中的 FDTD 公式在 [8] 中被考虑过。问题区域尺寸是$N_x \times N_y \times N_z$，其中$N_x$，$N_y$和$N_z$分别是$x$，$y$和$z$方向的上的元胞数目。如此一来，[8] 中给出的迭代磁场x方向的分量的方程就是

$$\begin{aligned} H_x^{n+1/2}(i,j,k) =& C_{hxh}(i,j,k)H_x^{n-1/2}(i,j,k) \\ &+C_{hxey}(i,j,k)(E_y^n(i,j,k+1)-E_y^n(i,j,k)) \\ &+C_{hxez}(i,j,k)(E_z^n(i,j+1,k)-E_z^n(i,j,k)). \end{aligned} \tag{A-1}$$

为了做到并行，线程被映射到元胞上去更新元胞。在映射中，一个线程块被构造为一个一维数列，如列表1中的第一列所示。另外在一维数列中线程被映射到如图1所示的 FDTD 域的 $x-y$ 平面中的元胞上。在 kernel 函数中，每一个线程被映射到一个元胞上，线程序号被映射到i和j上。然后，每个线程在z方向上在for循环中通过增加k序号的方式遍历元胞。对于每个k更新场值，因此整个 FDTD 域被遍历。

不幸的是在 FDTD 更新中内存读写起到支配作用。为了保证高性能所有的全局内存读写应被级联。通常，一个 FDTD 域的尺寸应该是任意数值。为了做到级联内存，FDTD 域要被通过填充x或者y方向上的元胞数目来使这个数目是16的整倍数，如图2所示。修正的 FDTD 域尺寸是$Nxx \times Nyy \times Nz$，其中Nxx，Nyy和Nz是分别是x，y和z方向的上的元胞数目。

在保证级联内存读写之后，数据for循环中的在z方向上的复用和适当使用共享内存也需要在 CUDA 代码中得到完善。代码的一部分如列表2所示。完善后的代码在一个 NVIDIA®Tesla$^{\text{TM}}$C1060 计算卡上得到测试。立体 FDTD 问题域的

尺寸递增，每秒处理的百万元胞数作为 CUDA 程序性能的测试。结果的分析如图3所示。可以看到代码每秒平均处理大约450百万个元胞。