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

With the widespread engineering applications ranging from broadband signals and non-linear systems, time-domain integral equations (TDIE) methods for analyzing transient electromagnetic scattering problems are becoming widely used nowadays. TDIE-based marching-on-in-time (MOT) scheme and its fast algorithm are researched in this dissertation, including the numerical techniques of MOT scheme, late-time stability of MOT scheme, and two-level PWTD-enhanced MOT scheme. The contents are divided into four parts shown as follows.

Keywords: time-domain electromagnetic scattering, time-domain integral equation (TDIE), marching-on in-time (MOT) scheme, late-time instability, plane wave time-domain (PWTD) algorithm

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 sub-area, 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

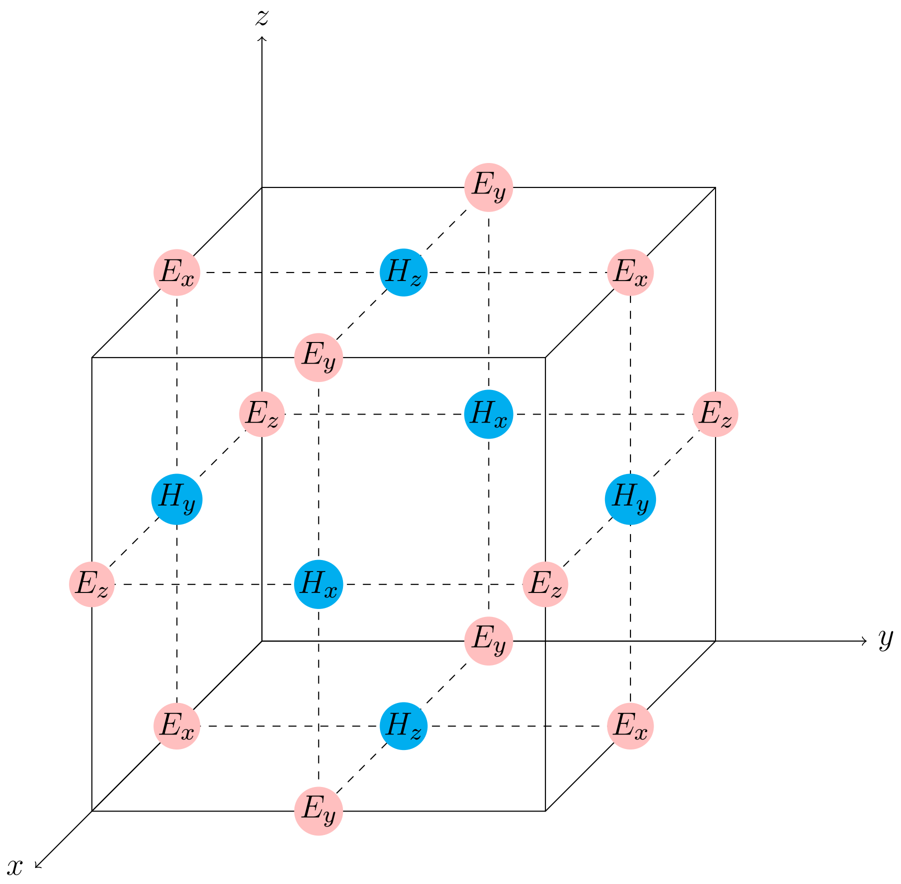


图 2-1 The spatial discrete structure of Yee cell value of magnetic field of the last time. Iterating the process in a marching-in-time way, we can make an analog to the continuous electromagnetic waves's propagation.

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	n
E_y	i	$j+\frac{1}{2}$	k	
E_z	i	j	$k+\frac{1}{2}$	
H_x	i	$j+\frac{1}{2}$	$k+\frac{1}{2}$	$n+\frac{1}{2}$
H_y	$i+\frac{1}{2}$	j	$k+\frac{1}{2}$	
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 princess 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.

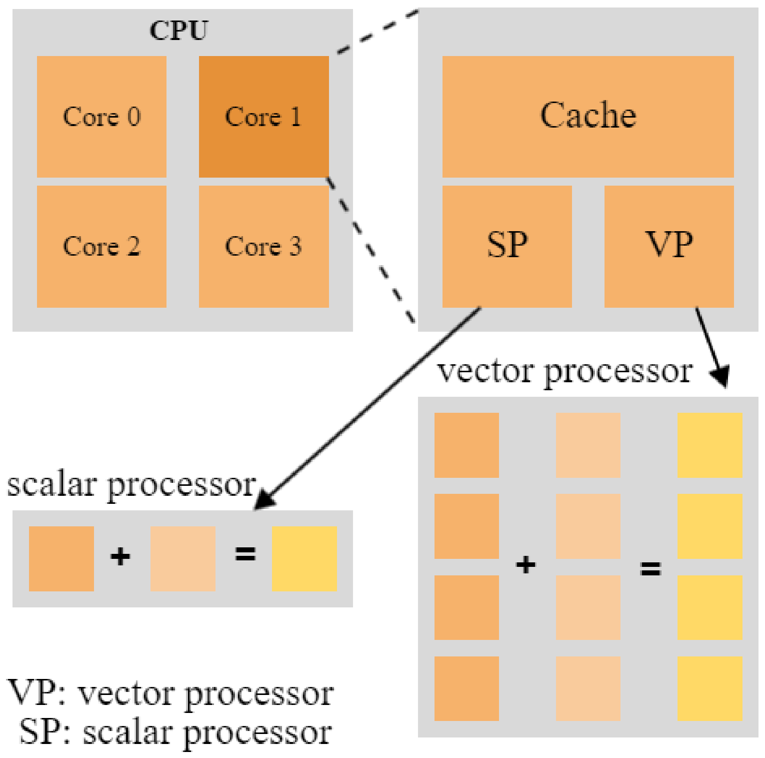


图 3-1 Vector processor and scalar processor

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

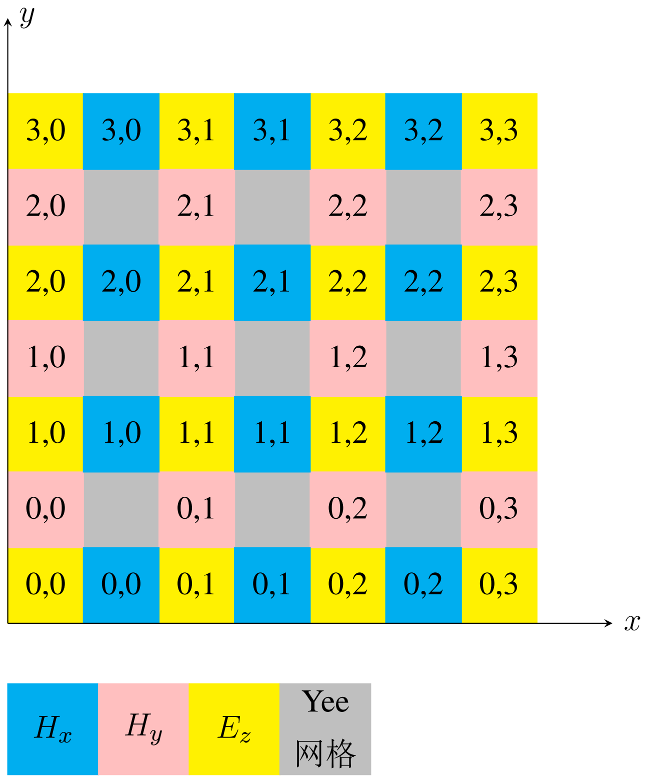


图 3-2 The position of discrete field points in Yee cells of TM mode wave

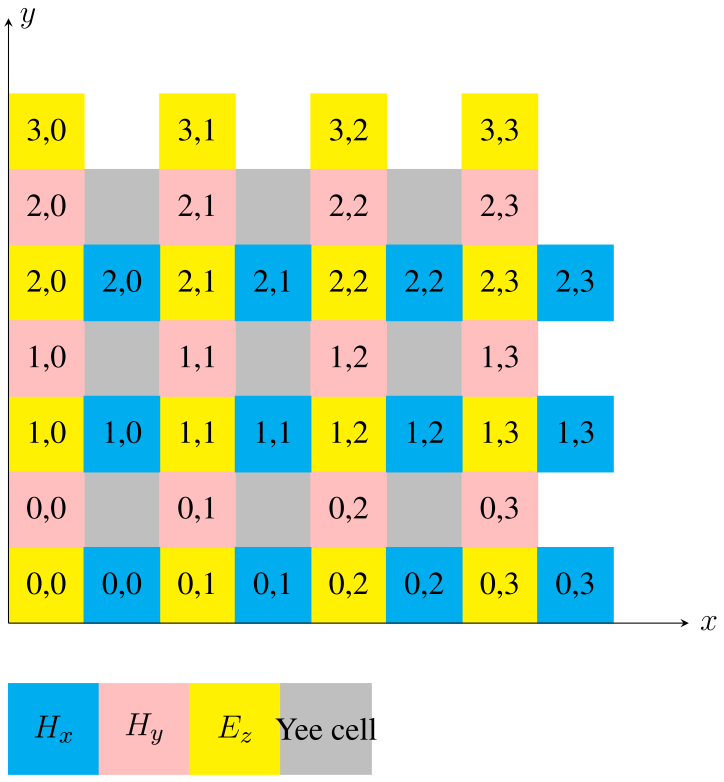


图 3-3 The computational model of traditional data parallelism method

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$\times$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 $V_{E_z}^1$. 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 $V_{E_z}^2$.

The second step, we use instructions to have $V_{E_z}^1$ minus $V_{E_z}^2$, and note the result as V_{sub}. More specifically speaking, let the first data of the $V_{E_z}^1$, minus the first data of $V_{E_z}^2$, 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.

$$\begin{array}{ccc} V_{E_z}^1 & & V_{E_z}^2 \\ E_z(0,0) & - & E_z(0,1) \rightarrow H_x(0,0) \\ E_z(0,1) & - & E_z(0,2) \rightarrow H_x(0,1) \\ E_z(0,2) & - & E_z(0,3) \rightarrow H_x(0,2) \\ E_z(0,3) & - & E_z(1,0) \rightarrow H_x(0,3) \\ E_z(1,0) & - & E_z(1,1) \rightarrow H_x(1,0) \end{array}$$

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.

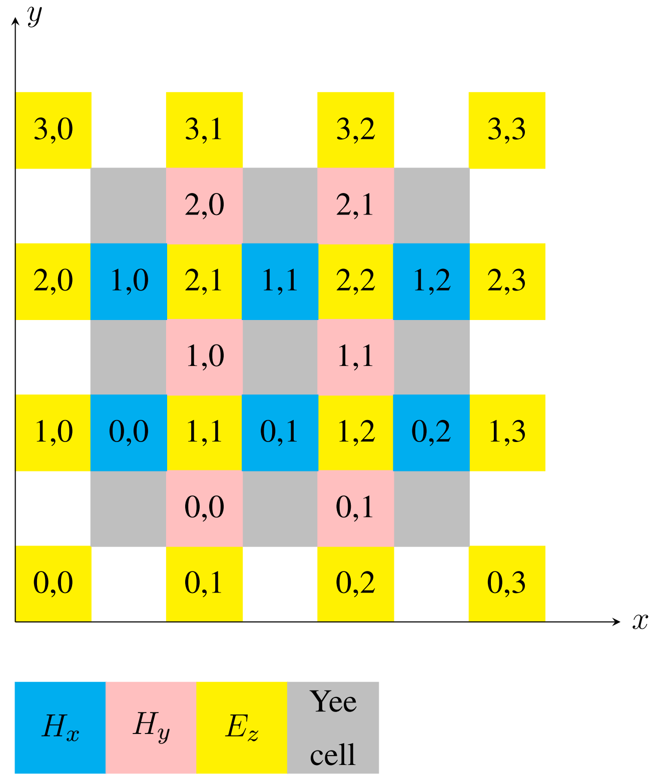


图 3-5 The modified computational model of data parallelism

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 $E_z(2\text{fi}0)$ 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:

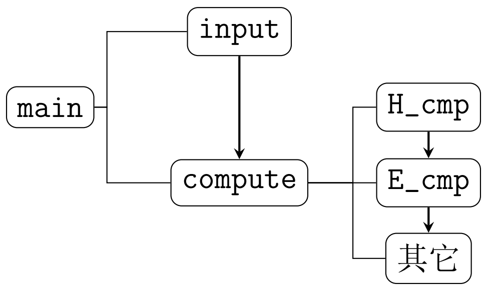


图 3-6 The framework of the program

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$\times$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 使用图形处理器对FDTD加速

在本章中，我们以二维TM波为例来展示使用图像处理器的加速方案。

4.1 图形处理器与CUDA

图形处理单元（Graphic Processing Unit，GPU，以下简称为GPU），是一种专门运行绘图运算工作的微处理器，于1999年8月被 NVIDIA 公司提出，自此作为一个独立的运算单元。早期的 GPU 是随着图形界面操作系统的的普及而得到推广。这是 GPU 提供基于硬件的位图运算功能。同一时期，在专业计算领域，Silicon Graphics 公司一直致力于推动3D图形的应用，并于1992年发布了 OpenGL 库，试图将其作为一种与平台无关的标准化3D 图形应用程序编写方法。

2001年，NVIDAI 公司开发了 GeForce3，它拥有可编程功能，同时支持 OpenGL 和 DirectX8，宣告了 GPU 并行计算的开始，是 GPU 技术上最重要的突破。开发人员也是从此时开始第一次可以对 GPU 中的计算机型一定程度上的控制。因为GPU的主要目标是通过可编程计算单元为屏幕上的每个像素计算出一个颜色值，虽然通常这些计算单元得到的信息是图像的位置、颜色等等，但是实际上是可控的，这吸引了许多人员来探索在图形渲染之外的领域中利用GPU。此时的的GPU计算使用起来非常复杂。标准图形接口，例如 OpenGL 和 DirectX 是和 GPU 交互的唯一方式，因此使用 GPU 就不得不利用图形 API 的编程模型。这样一来，人们只能按照图形计算的方式，将其它任务按照图形渲染任务的形式输入 GPU，然后得到 GPU 返回的图像格式的结果。这种方法可行，并且极大地提升了计算任务的完成效率，但是太过复杂，需要人们掌握图形相关的知识，对研究人员负担过重，并有很大局限。例如对写入内存的限制，以及无法控制是否能处理浮点数，在程序出现问题的时候也没有办法进行调试。

2006年，NVIDIA 发布了 GeForece8800GTX，是具有里程碑意义的一步。从这里开始，GPU 第一次设计了适合于通用计算的架构，包括包括了线程通信机制、多级分层存储器结构以及符合 IEEE 标准的单精度浮点运算和逻辑运算的结构，使其拥有了强大的并行处理能力。

2007年6月，统一设备计算架构（Compute Unified Device Architecture，CUDA。以下简称为 CUDA）正式开始发布，这引起了 GPU 通用计算的革命。CUDA 统一了之前 GPU 中的计算资源，使得执行通用计算的程序能够对芯片上的每个数学逻辑单元进行排列，并且这些逻辑单元都满足 IEEE 单精度浮点数学运算要求。此外，还可以任意的读/写内存，以及访问共享内存。在使用方面，CUDA 采用了工业标准 C 语言，只增加了一部分关键字来支持 CUDA 的特殊功能。大大减轻了使用者的学习负担。

4.2 使用图形处理器加速的原理

4.2.1 CUDA编程模型

在使用 GPU 的时候，由于所用的 GPU 是外接设备，所以 GPU 和内存和 CPU 是相互独立的。习惯上我们称 CPU 为 host，GPU 为 device。Host端的代码主要负责执行串行部分，主要是负责整个计算的流程的调控。Device 端的代码负责具体计算。在运行时，我们是在 CPU 中开始运行程序，在需要 GPU 的运算时，在程序中声明一个 kernel，来调用 GPU 进行计算。计算开始时，由于内存系统是相互独立的，所以我们将数据传输到 GPU 中进行计算，计算后再传输回 CPU 继续进行程序的运行。整个过程如图4-1所示。

接下来我用将用一个如下所示的矢量求和代码来简要介绍CUDA编程模型：

```
#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;
}
```

我们看到，首先我们需要使用修饰符 __global__ 来表明接下来的函数是一个 kernel，意味着该函数将在 device 上被执行，且该函数可以被 host 以及 device 上的函数所调用。在 host 中调用 kernel 需要以尖括号的形式通知 device 使用什么方式运行 kernel。尖括号中第一个参数是表示设备在执行核函数时使用的并行线程块的数量，第二个参数表示每个线程块中启动的线程数目。在使用 kernel 之前，我们需要调用 cudaMalloc() 在 device 上为三个数组分配内存，在使用完之后使用 cudaFree() 释放内存。之前我们提到 device 和 host 的内存是相互独立的，因此我们在计算前需要在 host 中设置好数据的初始化，将数据使用 cudaMemcpy() 输入到设备当中，在 device 完成计算之后同样使用该函数将计算结果传递回 host。

4.2.2 线程

并行运算的基本单位单位是线程。在 GPU 中，线程有三个层次。最底层的是线程，由线程构建出线程块，多个线程块又组成线程格。一个线程格对应一个 Kernel。目前的设备中，一般可以调用最多65536个线程块，每个线程块可以调用最多512个或1024个线程，线程块内的线程可以共享内存，快速交换数据。在这里的示例中，作为 kernel 的 add() 函数调用了128个线程块，每个线程块使用了128线程。这时我们可理解为，运行时 device 将会创建函数 add() 的128×128个副本，并以并行的方式同时计算每个副本。

在示例代码的第7行，我们看到有变量threadIdx.x、blockId.x和 blockDim .x。这些变量都是 CUDA 的内置变量，分别代表每个线程的编号、每个线程格的编号和线程格中每一维的线程数量。这些变量在运行时已经预先定义，变量中包含的值就是当前执行 device 代码的线程索引。对于每一个线程或者线程格而言编号都是不同的，对于线程格而言，每一维的线程数量是固定的。线程格最高支持三个维度，下图4-2展示了二维线程块和线程的关系作为示例。

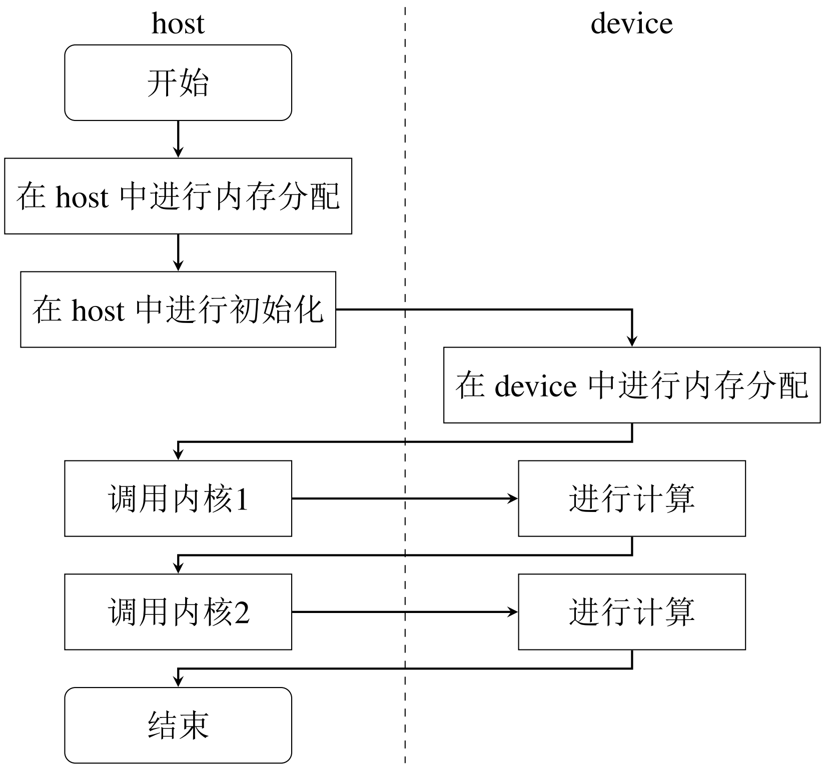


图 4-1 使用 CUDA 计算流程

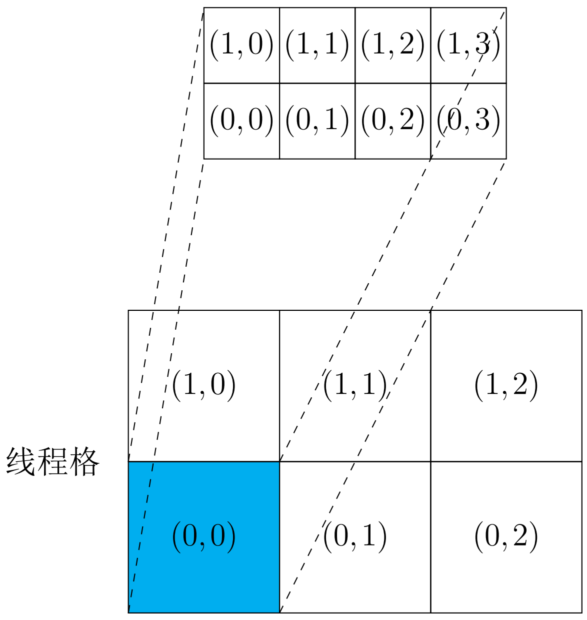


图 4-2 线程格和线程的二维层次结构示意图

根据这个关系，我们可以计算出利用这种关系得出一组唯一的索引，即如示例代码中第7行所示`tid=threadIdx.x + blockId.x * blockDim.x`。利用这个索引我们可以对数组中的元素进行一次的不重复的计算，当一个线程完成对当前元素的计算的时候，将当前索引加上全部线程数目，如示例代码的第9行所示，进行对下一批的元素的计算。这样就完成了对任意数量的数据的计算。

4.3 FDTD的CUDA加速方案

针对使用 CUDA 对 FDTD 加速的问题，我们采取的策略为以一维线程格作为 Yee 网格排布中的行，以每个内存格中的一维线程作为 Yee 网格排布中的列。在使用该策略进行计算之前，我们需要对其它部分进行优化，以便提高效率。

首先是传输优化。为了对某个数据集进行操作，如前文所说需要在 CPU 中对数据集进行初始化，然后将数据从 host 传输到 device 中，然后在对数据进行操作之后传输回 host。这种做法将带来两个问题。

第一是由于传输过程是在完全串行的方式进行的，因此传输效率较低，尤其考虑到使用 GPU 加速的时候往往是要对一个较为巨大的数据集进行处理，因此将会有较长一段时间内 host 和 device 的内存都是闲置的，是对计算资源的浪费。第二个问题是 I/O 硬件吞吐量。对绝大多数程序而言，这是无法加速的限制。

针对这个问题，我们采取的措施是取消从 host 传输数据到 device 的步骤，转而在 device 上分配内存的时候直接初始化各个数据为0。这样一来，减少了一半数据传输的时间。

其次是内存优化。由于我们是以二维的 TM 波作为示例，因此各个场分量的节点将以二维形式进行排布。在此时我们采用 `cudaMallocPitch`取代`cudaMalloc`进行对 device 上变量的内存分配。在第二章中我们已经提到了内存对齐的问题，当内存不对齐的时候处理器读取地址的操作往往要耗费三倍的时间。当内存对齐之后，相比不对齐性能提高最大有50%。相对应的，我们使用`cudaMemcpy2D`取代`cudaMemcpy`将数据集传输回 host。

在上述优化的条件下，我们规划了具体的对 FDTD 加速的方案。程序整体架构如图4-3所示。程序分为两大部分，一部分是类，一部分是操作函数。我们定义了三个类。类 E 中包含了电场分量的信息，以及对电场分量的初始化、检查信息和写入文件的几个方法。类 H 和类 E 相似，针对 H_x 和 H_y 两个场分量做了同样的规定。类 src 针对激励源包含了激励源以及仿真环境的信息，以及随时间激励

源变化的方法。操作函数负责仿真的核心计算，可以分为电场分量的计算、磁场分量的计算以及对边界条件的处理三个部分。

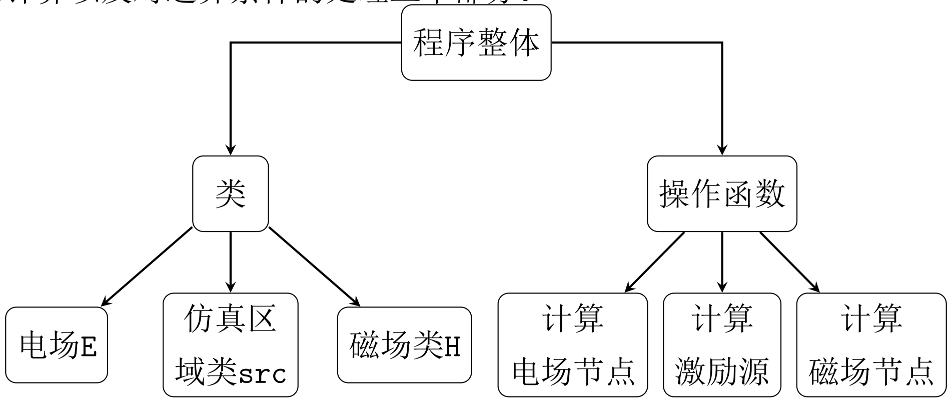


图 4-3 使用 CUDA 加速的程序的组成架构

程序的运行流程如图4-4所示，首先对仿真环境进行初始化，然后对各个场分量进行初始化，接下来跳过传输 host 上数据到 device 的步骤直接开始计算。在计算后将数据传输回 host，写入文件。

在计算过程中，我们采取前文 CUDA 编程模型范例中的并行方法，使用每个线程计算每个场分量节点。以对电场节点的计算函数为例，核心代码如下：

```
__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
```

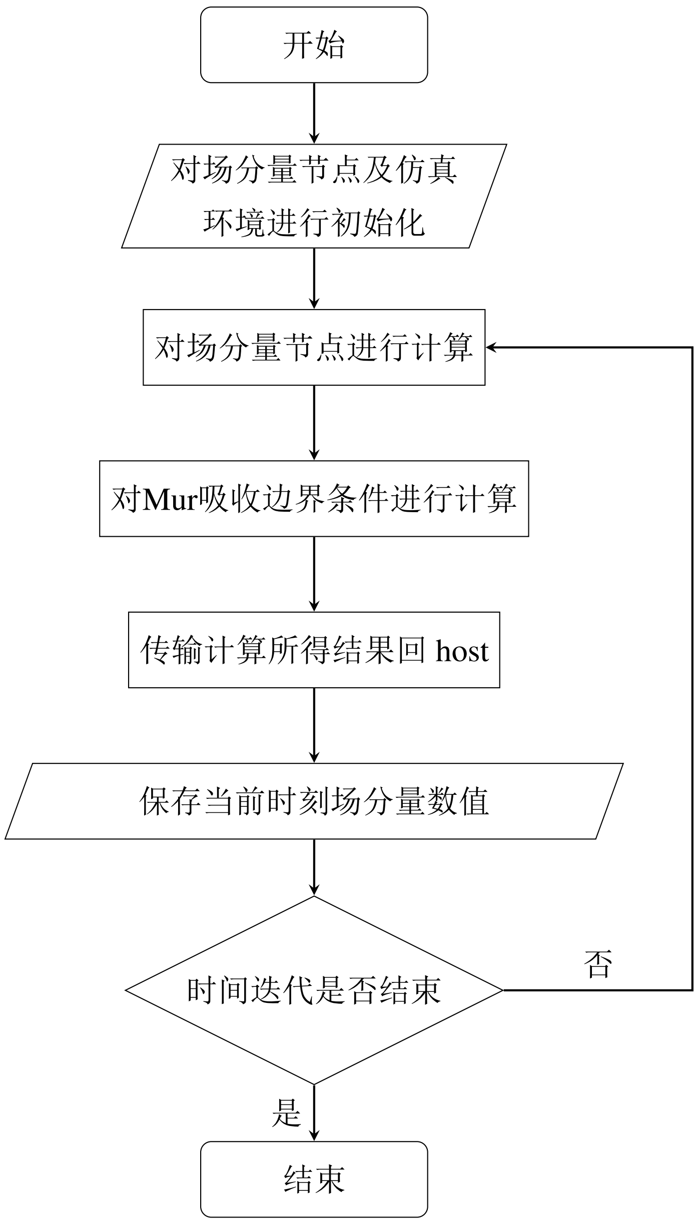


图 4-4 使用 CUDA 加速的 FDTD 程序流程图

```
        //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 使用CUDA加速后性能提升情况

此处同样使用二维 TM 波 FDTD 的 Mur 吸收边界条件为例来比较使用 CUDA 加速和不使用任何加速、使用 CUDA 加速和使用矢量处理器加速以及不同时间或空间规模下的 CUDA 加速情况对比。

程序的框架以调用树的如图4-5所示。在分析时，我们采用和第三章第四节中的分析方法。下面各表展示了不同样本下的性能对比结果。

表4-1展示了传统串行计算和使用 CUDA 加速的的性能比较。我们可以看到，在每次对计算函数的调用中，在串行计算中计算场分量的函数（H_cmp和E_cmp）在计算中花费了大量时间，但是在 CUDA 加速方案中花费了不到1%的时间就完成了计算。事实上，在 CUDA 加速方案中，在每一次时间迭代中，对于各个场分量的每一个节点都是一个个独立的线程同时计算的。所以事实上 CUDA 方案中在计算上花费的时间理论上只有传统串行方案的$1/(2000*1000)$。在性能分析报告中我们可以看到 CUDA 方案中计算函数的占用时间总共只有程序运行总时间的0.81%。绝大部分时间都是被各个场分量的初始化所占用，总占用在90%以上。其中对H_x分量分配内存的cudaMallocPitch函数占用了程序运行总时间的89.3%，还未探明原因所在。这也体现了 CUDA 加速方案的一个限制，就是由于 CUDA 是一个独立设备，所以在调用 CUDA 中会出现通信、内存方面的许多注意事项。

在表4-2中展示了 CUDA 加速和数据并行加速的性能对比。通过对比我们发现 CUDA 加速方案中对于场分量节点的计算用时仍然不到数据并行方案用时的1%。但是我们发现尽管计算时间很短，但是整个程序的运行时间很长。在 CUDA 方案

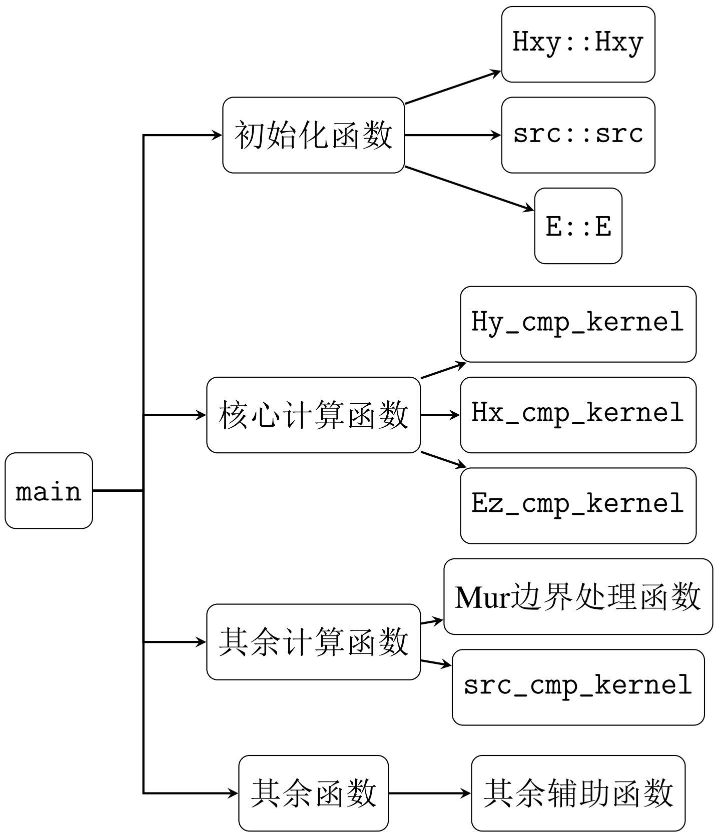


图 4-5 CUDA 加速的 FDTD 程序调用树

表 4-1 传统串行方案和 CUDA 加速方案性能比较

函数	已用非独占时间（ms）		使用 CUDA
	传统串行方案	CUDA 加速方案	所节约时间
$main$	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] 仿真空间大小为2000 × 1000个 Yee 网格。

[2] 时间迭代次数为1000次。

[3] 仅串行方案中。

[4] 仅 CUDA 加速方案中。

中的`main`函数所占用的6203.20ms的时间中，有5500ms的时间没有进行任何动作。目前尚不清楚是因为什么原因引起的。

在表4-3中我们可以看到，除去1000×1000空间规模情况下的未知原因引起的计算缓慢之外，在其余两种空间规模的情况中因为都是采用一个线程计算一个场分量节点，因此对全部节点的计算时间非常小，理论上是等于计算单个数据的时间，和空间规模无关，在实际运行中我们无法对如此细小的时间进行测量。即使是在1000×1000空间规模的情况中，我们也可以看到计算所占用时间依然非常的少，在表4-2中我们已经看到，即使是数据并行，计算所需时间仍然在百倍以上。

在表4-4中我们可以看到，在不同时间规模下使用 CUDA 加速方案的情况下计算函数所占用时间依然极少。两个时间规模下程序仿真所占用时间的差别根据检测报告表可以看出是因为`cudaMallocPitch`在分配H_x场分量时所占用的时间差别所导致。在不同的方案中，该函数所占用时间独占时间均接近程序运行总时间的90%。NVIDIA 公司在关于该函数方面没有披露更多细节，我们暂时还无法得知为何该函数在第一次被调用时要使用如此多的时间。

4.5 结论

在本节中，我们介绍了使用 CUDA 加速的原理和编程模型，探讨了如何使用 CUDA 实现对 FDTD 方案的加速。最后，我们实现了二维情况下使用 CUDA 加速 FDTD，通过对仿真程序的性能测试，我们分析了使用 CUDA 加速的性能提升，并与传统串行方案以及上一节提到的改进数据并行计算模型进行比较。我们通过程序性能数据的分析我们可以看到使用 CUDA 加速对 FDTD 性能的巨大提升。但是在某些情况下出现的尚不清楚的在计算之外的大量时间占用情况也表示在使用 CUDA 加速时需要对程序进行更谨慎的设计，因为 CUDA 作为独立设备，同时有与 CPU 不同的架构，因此我们需要针对 GPU 的情况重新对 device 部分代码进行考虑。

表 4-2 CUDA 加速方案和改进数据并行方案性能对比

函数	已用非独占时间（ms）		使用 CUDA 所节约时间
	改进数据并行方案	CUDA 方案	
$main$	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] 仿真空间大小为1000×1000个Yee网格。

[2] 时间迭代次数为1000次。

[3] 仅串行方案中。

[4] 仅 CUDA 加速方案中

表 4-3 CUDA 加速在不同仿真空间规模下的性能

函数	已用非独占时间（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] 时间迭代次数为1000次。

表 4-4 CUDA 加速在不同仿真时间规模下的性能

函数	已用非独占时间（ms）	
	1000次时间迭代	3000次时间迭代
$main$	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 空间规模是2000×1000。

Chapter 5 结束语

本文针对 FDTD 算法的加速问题，探讨了使用 CPU 内置矢量处理器对其加速的方法，以及使用独立设备 GPU 对其加速的方法。针对两种方法，都给出了具体的实现方案，并以二维 TM 波的仿真为例进行了测试，对测试结果进行了分析与说明，得出了各个加速方案自身的加速效果。针对使用矢量处理器加速的情形，本文提出一种基于传统数据并行计算的改进方案，通过测试，改进方案比传统数据并行方案更有效。在使用外部设备，即 GPU 的情况下，在 CUDA 平台下的 GPU 对 FDTD 方法加速的效果卓越，因此本文认为，使用 GPU 进行计算是未来的趋势。

通过实例测试我们也看出来本文依然存在一些待解决的问题。其中最主要的是使用 CUDA 平台进行加速计算时对程序更为谨慎和全面的考虑。例如，使用常量内存来存储 FDTD 公式中的不变的参数而不是全局内存，以便减小内存带宽。在使用 CUDA 加速时，我们可以借鉴任务并行的 FDTD 算法，即将仿真区域分割成多个子区域，每个区域分派给一个独立的 CUDA 设备，多个 CUDA 设备协作计算。

在本次课题的研究中，在知识能力方面，我深入了解了 FDTD 算法以及该算法的多种边界条件。学习了 CUDA 平台的框架及其编程模型。在探究能力方面，学会了独立的收集文献，整理文献，在前人的科研成果上做进一步的研究。了解了基本的科研方法和思路。

Acknowledgements

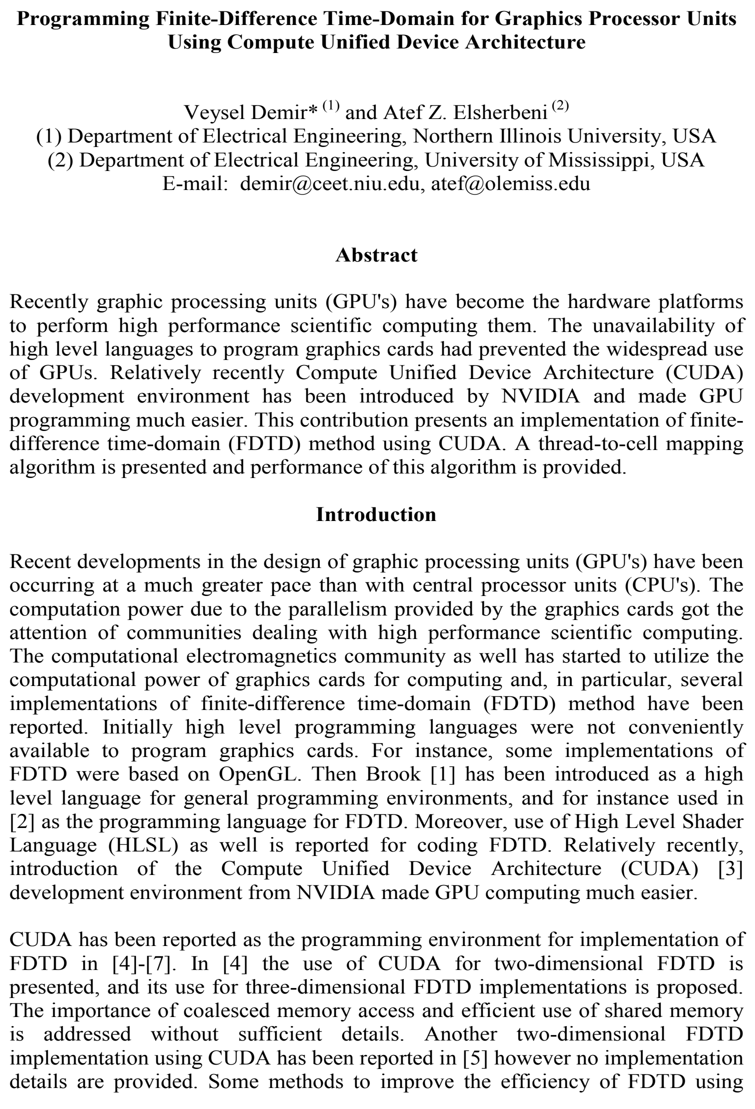
在攻读博士学位期间，首先衷心感谢我的导师 XXX 教授，……
……

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

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

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.

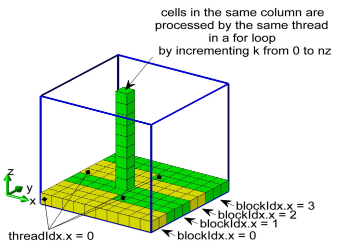


Figure 1. Mapping of threads to cells of an FDTD domain.

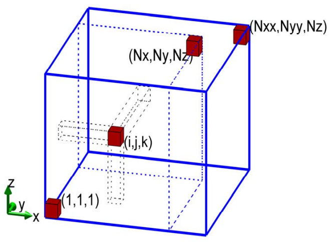


Figure 2. An FDTD problem space padded with additional cells.

```
__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.

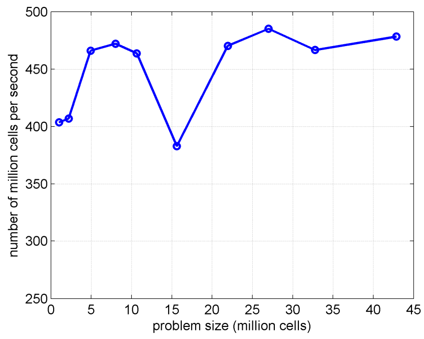


Figure 3. FDTD algorithm speed versus problem size.

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。