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

目 录

第1章 引言 1
第2章 FDTD理论 4
2.1 Yee元胞 4
2.2 Yee元胞的Maxwell离散形式 7
2.3 Courant稳定性条件 8
2.4 空间间隔的要求 10
2.5 边界条件 10
2.6 并行计算 10
2.7 本章总结 11
第3章 使用矢量处理器对FDTD进行加速 12
3.1 矢量处理器和数据并行 12
3.2 使用矢量处理器的FDTD加速原理 13
3.3 一种新的数据并行FDTD计算模型 16
3.4 两种计算模型对比 19
3.5 结论 21
第4章 使用图形处理器对FDTD加速 23
4.1 图形处理器与CUDA 23
4.2 使用图形处理器加速的原理 24
4.2.1 CUDA编程模型 24
4.2.2 线程 26
4.3 FDTD的CUDA加速方案 28
4.4 使用CUDA加速后性能提升情况 31
4.5 结论 33
第5章 结束语 36
参考文献 37
致 谢 38
外文资料原文 39
外文资料译文 44

第1章 引言

自从1873年 Maxwell方程建立以来，电磁波理论和应用在一百多年来取得了巨大的发展。电磁波的研究与应用已经深入到各个领域，例如雷达技术、微波、天线、无线电传播、博导建模、电磁成像、地下电磁探测，等等。在实际环境中，电磁波的传播过程非常复杂，例如复杂目标的散射，复杂的城市地形中的实际通信，波导和微带结构中的传播。为了具体的研究这些环境中电磁波的特性十分有用，实验和理论分析计算是两个重要的手段，通常只有一些经典问题有解析解，在结合实际环境的电磁参数进行计算时往往会遇到环境太过复杂无法得到解析解的情形，因此需要通过数值解得到具体环境下的电磁波特性，计算电磁学因此而发展起来。随着计算机技术的发展，目前已经提出求解麦克斯韦方程的许多重要的数值解方法，例如矩量法（MoM）、有限元法（FEM）、边界元法（BEM）以及时域有限差分法（FDTD）等。

1966年 K. S. Yee [1] 首次提出了 FDTD 方法，该方法在空间和时间上对麦克斯韦方程使用中心差分进行了离散，然后对得到的离散方程在软件或者硬件上使用“蛙跳”方式进行求解：电场分量在空间中就给定的某一时刻求解，然后磁场分量在同样的空间区域在紧接着的下一时刻进行求解，然后不断重复这一过程，直到电磁场演化到欲求的结果或者稳定。这一方法自提出后得到了迅速发展和广泛应用，在许多科学和实际问题，例如辐射天线的分析、微波器件和电磁兼容分析的解决中起到重要作用。

FDTD 的发展和应用面临两个主要问题，第一是边界条件问题。因为计算机容量的限制，FDTD 计算只能在有限区域进行，不可能计算开区域的电磁场。如果要模拟开域电磁过程，在计算区域的阶段边界处必须给出吸收边界条件。针对这个问题， Taylor 等 [2] 于1969年提出简单插值边界， Mur [3] 于1981年提出了一种十分有效的 Mur 吸收边界条件后来被广泛采用， Berenger [4] 于1994年提出了一种全新的吸收边界，即完全匹配层（Perfectly Matched Layer，PML），在此基础上 Sacks [5] 等和 Gedney [6] 等提出了各向异性介质的 PML， Chew 以及 Weedon [7] 提出坐标伸缩的 PML。这几种 PML 已经作为吸收边界得到了广泛的应用。

由于 FDTD 方法在使用时需要对整个计算区域进行格划分，并且由于 Courant 稳定性条件空间的格划分需要足够精细来求解最小的电磁波波长部分，所以在求

解较为大型的仿真区域时会产生很大的计算区域。时间步长和空间步长一样需要满足 Courant 稳定性条件，所以电磁场演进的时间步数目有最小值限制。这两点导致使用 FDTD 求解时需要很长的求解时间。因此 FDTD 方法面临的第二个问题就是缩短计算时间。 FDTD 具有天然的并行特点，即在计算时，两个相邻的子计算域只需要交换域交界面上的切向场值数据。利用这个特点，发展起了许多区域分割并行 FDTD 算法. 在这些并行算法中，原始待求解问题空间被分割为一些子空间，然后把每一个子空间分配给一个计算核心，在每一步的计算后都交换相邻子空间交界面上的数据。

区域分割并行算法是属于任务并行层次的并行算法。任务并行是一种较高层次的并行，更低一层的指令并行由 CPU 自发完成不需要人为参与，而最低层次的数据并行，则存在着待挖掘的计算性能。起初人们直接使用 CPU 计算时，使用的是 CPU 的算数逻辑单元（arithmetic logic unit， ALU），一次只能对单个数据完成一次运算。为了实现数据并行，挖掘 CPU 的计算潜力，我们需要使用 VALU 进行计算。 VALU 是一个中央处理单元，和 ALU 的不同之处在于在其中我们可以使用指令来对称作向量的一个数列进行操作，而不仅仅是单个数据，因此 VALU 能极大的提高计算性能，尤其是在数值仿真任务中。目前，在几乎绝大数商业 CPU ，例如 Intel 和 AMD ，都提供操作向量的指令集，例如 VIS ， MX ， SSE 和 AVX 。因此 VALU 的一大优势就是使用便捷。在过去的研究中， L. Zhang 和 W. Yu [8] 使用 SSE 操作 VALU 对 3D FDTD 的单精度计算进行了加速。 M. Livesey、 F. Costen 和 X. Yang [9] 将工作拓展到双精度计算中。 Y. L [10] 等给出了使用VALU计算的伪代码。然而之前的这些研究忽略了在不同边界条件的特征的差别。本文针对某些特定的边界条件，例如吸收边界条件（MUR），给出使用 VALU 计算 FDTD 时的新的计算模型，进一步挖掘计算效率的潜力。在文中，我们给出了 2D FDTD 的 C++ 实现作为示例，并通过 Visual Studio 对计算性能进行比较和分析。改计算模型可拓展到 3D FDTD 情形中。

虽然的 CPU 的各个并行层次都有了针对仿真计算的并行加速方案，但是 CPU 的设计目的是以更高的操作频率、更多的寄存器以及更复杂的 ALU 来作为计算机的核心来完成可能遇到的各种通用操作，而非专门完成数据的数学计算。因此使用 CPU 进行 FDTD 运算虽然是可行的，但是既不能充分发挥 CPU 的特点，也不能满足 FDTD 运算的效率需求。所以具有几百个计算核心、适合进行大量数据、重复计算的图形处理器（Graphic Processor Unit，GPU）开始被运用于 FDTD 计算。 GPU 的概念于1999年被提出，2004年，Sean E. Krakiwsky 等[11]首次尝试用

GPU 加速 FDTD 算法。由于起初 GPU 并不是被设计用于普遍计算的，因此使用 GPU 计算并不便捷，需要学习 GPU 的内部结构以及专门的硬件语言。这种情况在2006年产生了具有通用计算架构的 GPU 之后得到了改变。2007年， CUDA 正式发布。 CUDA 的出现允许人们方便的使用已经熟悉的 Fortran 、 C 等进行 GPU 计算编程，大大的加快了利用 CUDA 对 FDTD 进行加速的研究。 J. A. Roden 和 S. Gedney[12] 提出了 CPML 边界条件下使用 CUDA 加速的方案，但没有提供实现细节。 Veysel Demir [13]提出了周期边界条件条件下使用 CUDA 加速的 FDTD 方案。本文使用 CUDA 对二维 FDTD 的截断边界条件、 Mur 吸收边界条件以及 CPML 吸收边界条件进行加速。

第2章 FDTD理论

2.1 Yee元胞

Maxwell 方程式支配宏观电磁现象的一组基本方程，这组方程既可以写成积分形式也可以写成微分形式，因此计算电磁学中也有根据积分形式发展而来的积分方程数值解方法和微分方程数值解方法。FDTD 是以由微分形式的麦克斯韦旋度方程出发进行差分离散从而得到的一组时域公式为基础的。

麦克斯韦旋度方程为

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

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

在直角坐标系中，(2-1)，(2-1)写为

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

以及

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

上面的两组六个方程式各场分量关于时间、空间的一阶偏导的方程组，并且各个场量相互嵌套。首先要对连续变量进行离散，包含时间离散和空间离散。令 $u(x,y,z,t)$ 代表 E 或 H 在直角坐标系中某一分量。在空间方面，我们假设在各方向上空间是均匀离散的，在 x、 y、 z 方向上离散的网格长度分别为 Δx、 Δy、 Δz，用 i、 j、 k 分别表示 x、 y、 z 方向上的网格标识。在时间方面，我们假设

在时间轴上就行均匀离散，离散的时间步长为 Δt，用字符 n 来表示时间 t 的时刻标识。这样，某一场分量咋看时间和空间域中的离散就可以用如下符号表示：

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

一阶偏导数的有限差分格式有多种，例如前向差分、后向差分、中心差分以及指数差分。这里我们选择具有二阶精度的中心差分法。所以，以 x 方向为例，场分量对空间一阶偏导的差分格式为

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

场分量对时间的一阶偏导的差分格式为

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

现在对(2-4)和(2-3)中的六个方程都进行了离散。接下来就要考虑在空间和时间中对这些离散节点的排布。

在空间中，FDTD 离散的电场和磁场各节点的空间排布如图2-1所示。这就是 Yee 提出的 Yee 元胞的空间离散结构。

由图可见，在 Yee 元胞中，E和H的各自三个场分量在 Yee 元胞表面上进行离散。电场分量E_x、E_y和E_z在 Yee 元胞的棱边中间，方向与棱边一致，磁场分量E_x、E_y和E_z在 Yee 元胞的表面的中心，其方向垂直元胞面。每一个电场分量周围都有四个电场分量环绕，同样，每个磁场分量周围由四个电场分量环绕。这种取样方式符合法拉第感应定律和安培环路定律的自然结构。同时这种结构也适合麦克斯韦方程的差分计算，能够恰当的描述电磁场的传播特性。

在时间上，电场和磁场相差半个时间步长，交替离散，从而可以在时间上迭代求解，而不需要进行矩阵求逆运算。

综合空间和时间离散排布结构来看，Yee 元胞可以自然的描述电磁之间相互作用的过程，从而给定相应电磁问题的初始值以及边界条件，就可以利用 FDTD 方法逐时间步推进的求得以后各个时刻空间电磁场的分布。

为了准确描述每一个场分量离散点的位置，我们需要对空间位置进行编号。根据上述各个场分量的空间排布，定义各个方向上的 Yee 元胞棱边为整数编号，棱边的中间位置为半整数编号。具体各个分量的取值如下表所述。

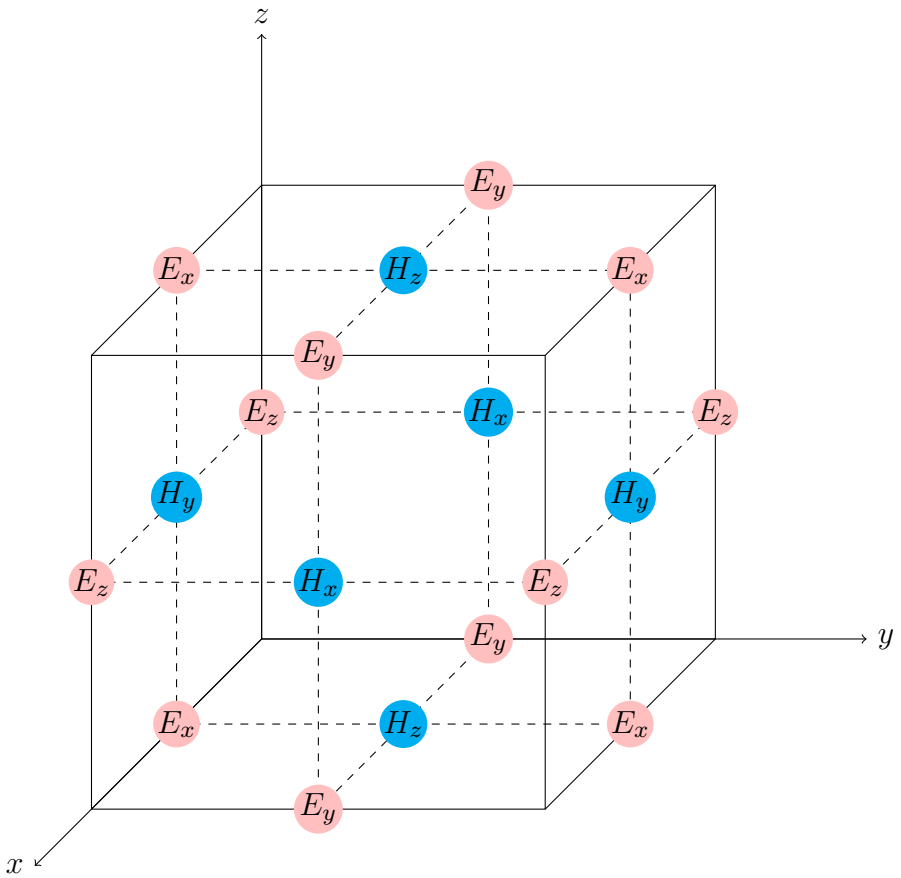

图 2-1 Yee元胞的空间离散结构

表 2-1 Yee元胞中E和H各分量节点的位置

场分量	空间分布位置			时间分布位置
	x	y	z	
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	

2.2 Yee元胞的Maxwell离散形式

我们以(2-3)的第一式为例来说明 Yee 元胞的 Maxwell 方程差分格式。根据(2-6)和(2-7)的差分格式，对(2-4)中第一式的各个部分写成离散格式，得到下式。

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

其中

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

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

m为等号左端场分量的坐标。本文中，为能更简洁说明方法起见，我们假设介质全为无耗介质，即$\sigma=\sigma(m)=0$，则有$CA(m)=1$，$CB(m)=\frac{\Delta t}{\varepsilon(m)}$，所以式(2-8)变为

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

按照同样的方法对H的场分量进行整理。以H_x为例，式(2-4)的第一式整理之后得到

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

其余的各个E和H的场分量的离散格式按照上述两式的方法整理，于是我们得到各个场分量关于时间迭代的离散格式。

2.3 Courant稳定性条件

根据上一节的内容，我们知道 FDTD 是用求解一组离散的有限差分方程来代替求解连续的偏微分方程组，即麦克斯韦旋度方程。所以和和许多数值方法相同，需要保证离散后的差分方程组的解释收敛和稳定的，否则是没有意义的。

首先考虑对时间间隔Δt的限制。由于任意场量在时域上可以分解为时谐场的叠加，所以考察对于时谐场的情况：

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

式(**??**)对时间的一阶偏微分为：

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

式(2-14)的左端用中心差分离散近似代替得到：

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

其中$u_n=u(x,y,z,n\Delta t)$。定义增长因子q为

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

将式(2-16)带入(2-15)得到方程

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

解方程，得到

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

如果要让随着时间步进的场量不发散，则需要让$|q| \leqslant 1$，由此得到

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

这就是一般场量对时间间隔Δt的稳定性要求。

面对包含6个场分量的Maxwell方程组，可以导出电磁场任意直角分量均满足齐次波动方程

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

由于任意波都可以展开为平面波谱，所以考虑平面波的解：

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

将(2-21)带入(2-20)并采用有限差分离散，得到

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

其中c为介质中的光速。该式整理并带入(2-19)后得到

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

该式的成立条件是

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

式(2-24)给出了空间和时间离散间隔之间应当满足的关系，称为 Courant 稳定性条件。

2.4 空间间隔的要求

在上一节中，时间间隔和空间间隔之间的关系得到了确定，接下来还需要确定空间间隔。以一维的情形为例。由(2-22)得到

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

我们看到，只有当$\Delta x\to 0$时，才能不出现色散，所以我们需要估计Δx小到什么程度可以接近0.根据三角函数，当$\phi\leqslant\pi/12$时，$sin\phi\approx\phi$，于是得到

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

这就是对空间离散间隔的要求。对于多维情形，各个维度的空间间隔同样满足关系式(2-26)即可。对于非时谐的宽带时域信号，只要空间离散间隔满足信号的最小波长可以满足关系式(2-26)即可。

2.5 边界条件

FDTD 方法把空间离散成一个个 Yee 元胞，然而实际上仿真的时候由于计算机资源的限制，不可能将无限空间仿真出来。另一方面，我们也只需要特定区域的电磁信息。基于这两点，我们需要在仿真空间边界处设定边界条件，以便在只计算仿真空间电磁信息的条件下得到更精确的实际情形中该区域的电磁信息。边界条件可以分为截断边界条件和吸收边界条件。在实际中应用的较为广泛的是吸收边界条件，其中又以 Mur 吸收边界条件和 PML 吸收边界条件为主。

2.6 并行计算

由于 Courant 稳定性条件，仿真计算区域的 Yee 网格必须分的足够细，这就限制 FDTD 在大型电磁计算问题中的应用。针对这一问题，并行计算是一个十分有效的解决方案。

计算机领域的并行计算一般分为三层。最上层是任务并行，即把需要计算机处理的任务切割为若干个子任务，将子任务分配给一些计算核心同时计算。在计算核心完成各自的计算之后，采用某种方法将各个子任务的计算结果组合起来得到完整的计算结果。由于FDTD的特点，即各个子区域在每个时间间隔中只需和

相邻的子区域交换交界处的数据，使得FDTD非常适合任务并行。针对这个层次的并行，目前有基于 MPI（Massage Passing Interface）和 PVM（Parallel Virtual Machine）这两种消息传递编程的方案。

并行的中层是指令并行，这个层次的并行大多数是由 CPU 制造厂商在开发 CPU 时处理的，FDTD 的并行计算基本不涉及这个层次的并行。

最底层的并行是数据并行。针对一些计算任务，在经过合适的规划后，可以在 CPU 执行一条指令时同时处理多个数据。在 FDTD 中的计算中，由于每个场分量的所有节点都有相同的计算公式，所以经过合适的安排每个场分量的数组，可以在每次计算时计算相邻的多个场点。针对这个层次的并行，也有了一些利用矢量处理器的加速方法。

2.7 本章总结

在这一章中，我们介绍了 FDTD 算法的原理以及 FDTD 离散形式，FDTD 方法的稳定性和收敛性和满足稳定、收敛的条件，FDTD 在仿真开区域电磁场时需要采用的边界条件，以及使用并行计算加速 FDTD 的方法。

第3章 使用矢量处理器对FDTD进行加速

在本章中，我们以二维TM波为例来展示使用矢量处理器的加速方案。

3.1 矢量处理器和数据并行

在实际中常常会有对多个数据执行相同的操作，例如两个向量的内积，称之为矢量操作。对单个数据的操作称为标量操作。需要注意的是，这里的矢量仅仅说明数据是多个，而非是指代物理上有大小和方向的量。此处与之对应的标量是指单个数据，矢量和标量侧重的是数据数量的单数与复数，它们各自的简化结构示意图如图3-1所示。数据并行就是对多个数据同时执行相同的操作，经典的使用数据并行的领域就是多媒体应用。矢量处理器的发展就是为了处理数据并行。

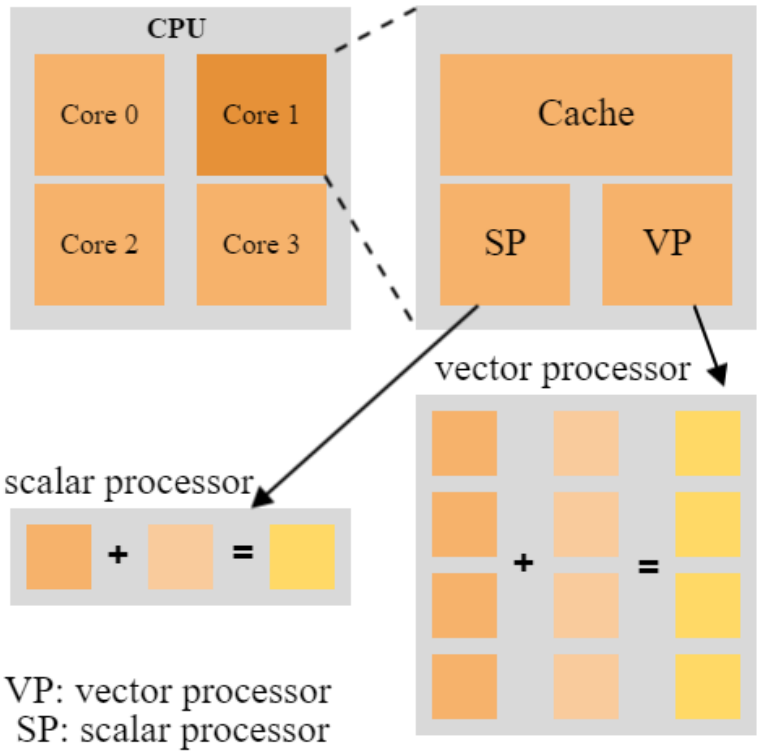

图 3-1 矢量处理器与标量处理器

矢量处理器是可以利用指令操作矢量的中央处理单元。在使用时，使用指令让矢量处理器读入一个或多个经过打包（packed）的多个数据形成的一维数据列，

然后使用指令让矢量处理器对这一个或者多个矢量的对应分量之间同时完成相同的操作，然后使用指令将计算结果返回内存中。

在使用矢量处理器计算的时候有一个和使用标量处理器很不同的地方在于数据的读入。在使用标量处理器的时候，由于每次只用从内存中读入单个数据，对单个数据 CPU 都会自动的处理好数据在内存中的对齐问题，所以不需要使用者有额外注意的事项，只要获得该数据的地址后直接读取即可。矢量处理器则不同，每次读入数据都需要读入连续的多个数据。这多个数据需要使用者在分配内存的时候显式的在内存中对齐，否则在读入数据时会消耗大量的资源。虽然只要数据是在内存中连续的，即使不对齐也可以读入到适量操作器中。但是即使是最优情况下非对齐的读入速度仍然比对齐情况慢了4.5倍。

3.2 使用矢量处理器的FDTD加速原理

以 TM 波为例。根据第二章可以得到H_x，H_y，E_z三个场分量的 FDTD 离散格式方程：

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

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

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

接下来以简单的3×3网格来直观的说明数据并行计算的原理。根据第二章的场分量在 Yee 元胞中的空间排布方法，得到在3×3的 Yee 网格中，场分量的排布应如图3-2所示。其中格子内的序号代表该节点的位置，序号的第一位代表y方向的位置，即所在行数，序号的第二位代表x方向的位置，即所在列数。

传统的数据并行方法对图3-3中的排布做了修改，在x增加方向的边界处增加了一列作为计算辅助而并无实际意义H_x节点。现在，设在x、y方向的Yee网格个数分别为N_x、N_y，则各个场分量在各个方向的节点数目表3-1所示。

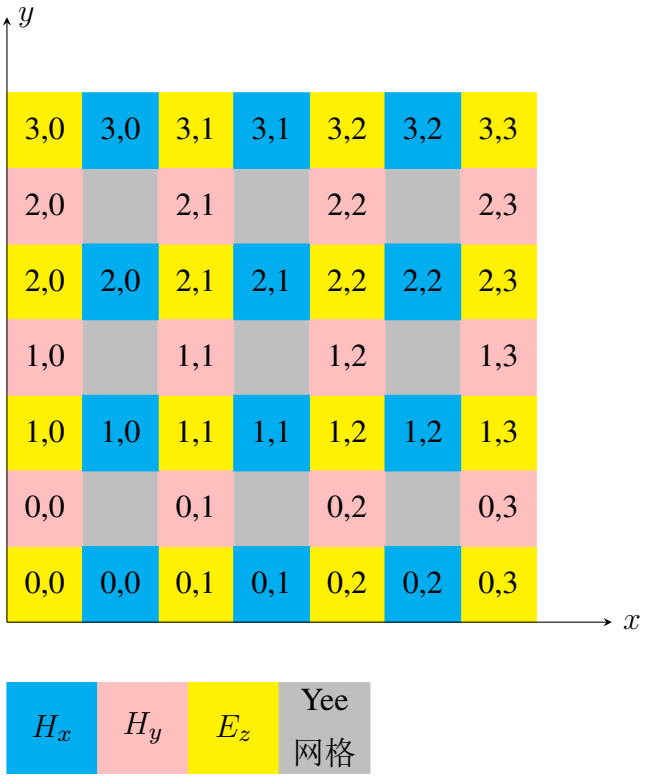

图 3-2 TM 波中各场分量的 Yee 网格排布模型

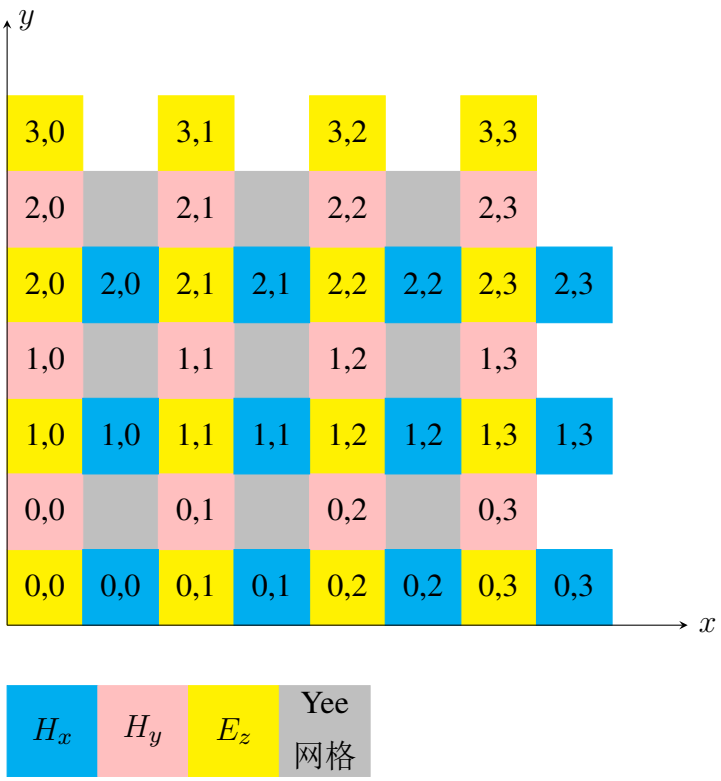

图 3-3 传统数据并行计算模型

在计算时，首先我们使用指令分别为三个场分量的各自相应大小的对齐的连续内存空间。在3×3的例子中，为E_z分配存放16个浮点数的内存空间，为H_x和H_y各自分配存放12个浮点数的内存空间。我们设定各个场分量都是先在x方向增长然后在y方向增长的，即节点的序号中首先增长代表x方向的那一位。所以，以E_z的内存举例，该场分量的内存中保存的节点按顺序分别为$(0,0)\cdots(0,3),(1,0)\cdots$。

在计算场分量时，以计算H_x为例说明，在说明示意时，不指定具体的操作适量操作器的指令集。第一步，根据(3-1)，我们读入三个矢量。第一个矢量包含H_x从第一个节点开始的五个节点，即$H_x(0,0)$到$H_x(1,0)$，记为V_{H_x}。第二个矢量包含E_z的从第一个节点开始的五个节点，即$E_z(0,0)$到$E_z(1,0)$，记为$V^1_{E_z}$。第三个矢量包含E_z从第二个节点开始的五个节点，即$E_z(0,1)$到$E_z(1,1)$，记为$V^2_{E_z}$。第二步，根据(3-1)，我们用指令让$V^1_{E_z}$减去$V^2_{E_z}$。减法操作为$V^1_{E_z}$的各项分别减去$V^2_{E_z}$中各自的对应项，得到的结果记为矢量V_{sub}。然后矢量V_{sub}的各项乘以常数$\frac{\Delta t}{\mu\Delta x}$得到记为$V'_{sub}$的结果矢量。接着用指令让$V'_{sub}$和$V_{H_x}$相加，得到最终结果。全部过程如图3-4所示。

$$V^1_{E_z} \qquad V^2_{E_z}$$

$$E_z(0,0) - E_z(0,1) \rightarrow H_x(0,0)$$

$$E_z(0,1) - E_z(0,2) \rightarrow H_x(0,1)$$

$$E_z(0,2) - E_z(0,3) \rightarrow H_x(0,2)$$

$$E_z(0,3) - E_z(1,0) \rightarrow H_x(0,3)$$

$$E_z(1,0) - E_z(1,1) \rightarrow H_x(1,0)$$

图 3-4 传统数据并行方案计算说明

我们看到增添的作为辅助的节点$H_x(0,3)$使得矢量处理器在计算同一个场分量时，读入的矢量节点序号遵从相同的相对关系。在刚才的例子中，改关系就是读入的两组E_z节点的序号分别是和H_x相同以及序号的两位同时增加1。如果去掉这些辅助计算点，首先是在矢量处理器读入节点的时候需要有额外的计算所需读入节点的序号，其次是对于每行节点，都会有小于等于4个节点需要额外采取标量计算的方式进行计算。这两点大大增加了程序的复杂性和运行效率。

3.3 一种新的数据并行FDTD计算模型

上一节提到的传统的 Yee 网格可以适用于各种边界条件。但是对于某些边界条件，例如 Mur 吸收边界条件和 PEC 边界条件，我们可以进一步的改进计算中 Yee 网格中场分量的空间排布模型，减少场分量的节点数目，从而减少计算量。首先我们先来看 PEC 和 Mur 边界条件。

PEC截断边界条件 PEC 边界条件中，设置边界上的切向电场分量为0，即

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

其中$E(k)$代表边界处电场节点。这种边界条件称为截断边界条件。

Mur吸收边界条件 使用 Mur 边界条件，就相当于在边界处涂吸波材料来吸收入射波，使得电磁波在边界处不会产生反射。考虑齐次波动方程的一维情形

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

解为

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

设置$x = 0$为左侧边界。由于仿真区域同时存在入射波和反射波，所以有

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

记该式右端第一项为u_-，第二项为u_+。对于边界$x = 0$而言，u_-为入射波，u_+为反射波。如果让电磁波在边界处不产生反射，就要将反射波变为0。因此得到

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

对该式进行差分处理，得到

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

其中$u(k)$代表边界点，$u(k-1)$代表边界内最接近的相邻点。

根据两个边界条件的公式(3-4)和(3-9)来看，在计算边界处的电场节点时不需要其它场分量的节点或者只需要最邻近处相同场分量的节点。因此，我们可以对之前的 Yee 元胞场分量排布的计算模型进行简化，去除边界处电场节点之间的磁场节点，从而减少所需计算的节点数目，减少计算量。我们仍然以3×3的规模的 Yee 网格为例来说明。新的计算模型如下图3-5所示。

表 3-1 TM波中各场分量在各方向上的节点个数

场分量	x方向未知量个数	y方向未知量个数
E_z	N_x+1	N_y+1
H_x	N_x+1	N_y
H_y	N_x+1	N_y

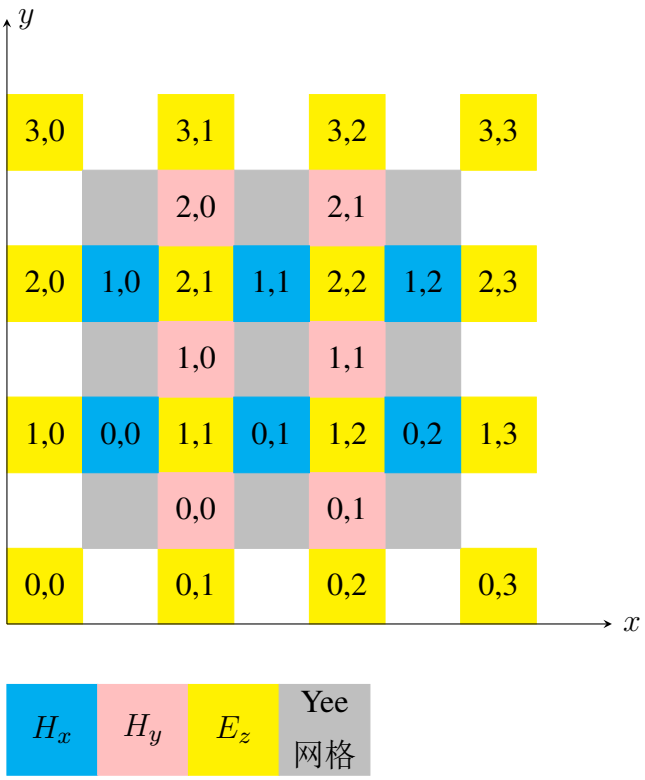

图 3-5 改进的数据并行计算模型

从图中我们可以看到，处于 Yee 元胞所包含的范围之外的磁场节点全部被简化掉。现在的 Yee 网格计算模型中各个场分量在各方向上的节点数目如表3-2所示。同样设在x、y方向的 Yee 网格个数分别为N_x、N_y。

表 3-2 TM 波中各场分量在新计算模型中各方向上的节点个数

场分量	x方向未知量个数	y方向未知量个数
E_z	N_x+1	N_y+1
H_x	N_x	N_y-1
H_y	N_x-1	N_y

在计算时，和上一节中传统计算方法相同。首先我们使用指令分别为三个场分量的各自相应大小的对齐的连续内存空间。在3×3的例子中，为E_z分配存放16个浮点数的内存空间，为H_x和H_y 各自分配存放6个浮点数的内存空间。我们同样设定各个场分量都是先在x方向增长然后在y方向增长的，即节点的序号中首先增长代表x方向的那一位。

上一节中的计算模型中，由于H_x的辅助节点的存在，使得三个场分量各自的内存中所保存数据的位置和该数据对应的节点的序号是一致的，即在内存中第m个数据对应的节点分别是$E_z(m_y,m_x)$、$H_x(m_y,m_x)$和$H_y(m_y,m_x)$，三个节点的序号相同。但是在改进的计算模型中，该关系不再存在。这就导致矢量处理器在计算某个场分量时，读入的各个矢量中所包含的节点序号不再遵从相同的对应关系。同样以H_x的计算为例来说明。按照上一节中的操作，在计算H_x的钱四个节点时，矢量处理器需要读入三个矢量，分别是包含H_x从第一个节点开始的前四个节点，即$H_x(0,0)$到$H_x(1,0)$，记为V_{H_x}。第二个矢量是包含E_z第二行第一个节点开始的四个节点，即$E_z(1,0)$到$E_z(1,3)$，记为$V^1_{E_z}$。第三个矢量是包含E_z从第二行第二个节点开始的四个节点，即$E_z(1,1)$到$E_z(2,0)$，记为$V^2_{E_z}$。按照上一节所述的计算步骤，应该如图**??**所示让$V^1_{E_z}$减去$V^2_{E_z}$。但是此时我们发现，两个矢量中前三个对应项的计算还正确，但是用来计算$H_x(1,0)$的第四项的减法就出现了错误。按照公式(3-1)，$H_x(1,0)$需要$E_z(2,1)$减去$E_z(2,0)$，但是实际上却是$E_z(2fi0)$减去$E_z(1,3)$。

这个改变之处，导致了我们在计算时需要做两点调整，第一是在设计节点序号的地方，包括矢量处理器读入节点数据和对场分量的节点的计算，不能像传统方法一样将一种算法应用在整个场分量的内存所保存的数据上，而是必须将场分量分段处理，往往是按照 Yee 网格的行数分段，对于每一段分别应用某种算法。

第二是在对场分量分段后每一段内存中的数据的处理，不能全部应用矢量处理器来计算。必须考虑到在每一段节点中，可能需要对该段前几个节点、中间部分节点和后几个节点分开处理，其中头尾处的节点需要使用标量处理器而非矢量处理器来计算。

3.4 两种计算模型对比

本文以二维 TM 波 FDTD 的 Mur 吸收边界条件为例来比较传统计算模型和改进计算模型的计算效率。我们分别对比使用传统计算模型和未使用数据并行的计算，以及在不同规模的仿真区域以及仿真时长下两种计算模型的区别。

程序的框架以调用树的形式展示，如图3-6所示矩形框代表函数，箭头代表执行顺序。我们可以看到，程序分为两大部分，分别是对数据进行预处理的 $input$ 部分和对场分量节点进行计算的 `compute` 部分。其中 `compute` 又分为对磁场节点进行计算的 `H_cmp` 部分和对电场节点进行计算的 `E_cmp` 部分，以及进行辅助工作的其它函数。

在分析时，我们采用 Visual Studio 的性能分析工具来收集样本的运行性能数据并进行分析。主要有两种分析方法，第一种是采样分析方法，第二种是检测分析方法。采样分析方法在每个 CPU 周期中断 CPU 并收集函数调用堆栈，对于不同类型的函数进行不同模式的样本数累加，从而收集分析运行期间应用程序所执行工作的相关统计数据。检测分析方法主要手机程序中函数调用的详细计时信息。该方法将捕获并检测每个函数的时间信息。时间信息主要包括两种，分别是：

已用非独占时间 执行某函数或代码行所用的总时间。

已用独占时间 执行某函数体或代码行所用的时间。不包括执行该函数或代码行所调用的函数所用的时间。

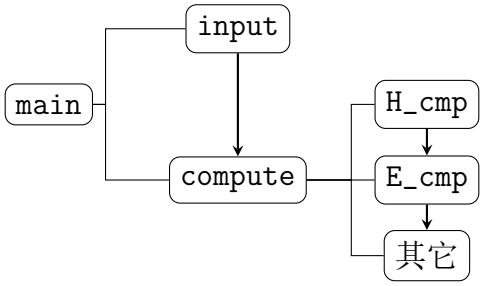

图 3-6 数据并行程序框架

所以，在样本程序中，main 函数的已用非独占时间就代表了程序总花费时间，compute 函数的已用非独占时间就代表了程序中计算场分量场点的时间花费。由于 H_cmp 和 E_cmp 函数没有调用函数，于是它们的已用独占时间和已用非独占时间相同，都是代表了执行一次该函数所用的时间。事实上，我们对这两个函数所关心的正是单次运行的时间。

为了保证各方法计算结果的准确性，各个样本的计算都运行5次去性能的平均值进行对比。编译时采取 Release 选项，这样能排除程序执行细节带来的对性能的影响，更能比较各个方案之间的性能差异。

计算环境为 Intel®Core™i5-3210M 2.3GHz，4.00GB内存，操作系统为64-bit Windows 10。

下列各表展示了不同样本下性能对比结果。

表 3-3 传统串行方案和改进计算模型方案性能比较

函数	已用非独占时间（ms）		使用数据并行所节约时间
	不使用数据并行	改进计算模型	
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] 仿真空间大小为2000×1000个Yee网格。

[2] 时间迭代次数为1000次。

根据表3-3所示，我们可以看出在不适用数据并行的样本中 H_cmp 和 E_cmp 1000迭代的总占用时间分别约为6160ms和4100ms，两者之和为10260ms，占用 main 函数已用非独占时间的约99.3%。同样，在使用数据并行的样本中该比例数值为99.2%。这意味着这两个函数是程序的关键函数，所以如果不同的样本性能有有效差异，那么我们可以肯定的说差异就是因为这两个函数的不同导致的。在表3-3的比较中，我们可以看到使用数据并行可以节约大约24%的时间，可以显著的缩短 FDTD 的计算耗时。

在表3-4中我们可以看到，和传统数据并行方案相比，新的数据模型可以节省 FDTD 计算时间约为 3.45%。基于上文对程序结构的分析，我们有理由排除掉程序细节的差异，得出改进后的计算模型优于旧方法的结论。我们可以看到，尽管删去一些磁场节点之后程序变得复杂，并且有一些节点的计算方式由之前的矢量

处理器计算转换为了使用标量处理器计算，从而造成了部分性能的下降，但是减少计算节点而带来的性能增加可以完全抵消掉这一部分的性能下降，并且能在总体上提升性能。

在表3-5中我们可以看到，仿真计算所需的时间和仿真区域的规模成线性关系，时间复杂度为$O(n)$，所以改进后的模型可以用在实际应用中。这里所说的仿真区域的规模，是指 Yee 网格的数目。根据实验所收集的数据，当仿真规模增长了n倍的时候，时间消耗是之前的$0.92n + 0.08$倍。

在表3-6中，我们可以看到随着时间迭代次数的增加，程序的仿真时间也随之进行按照线性变化。不过我们也注意到，计算场分量的两个函数H_cmp和E_cmp的平均时间随着时间步的增加在下降。但是下降并不明显。时间迭代次数增长到了原来的300%，时间减小幅度仅为1%。我们推测这一部分的下降仅仅是由于增加了时间迭代次数之后，计算机能更为准确的进行内存命中，从而减少时间消耗，和算法是无关的。

3.5 结论

在本章中，我们以二维的TM波仿真为例，来展示说明了 FDTD 的数据并行计算中的一种新的计算模型。该模型主要是在传统计算模型的基础上修改了 PEC 截断边界条件和 Mur 吸收边界条件的情况下在边界处的磁场场分量点的分布。通过削减部分的磁场场分量计算点，在不影响计算结果的情况下，减少了仿真所需计算量，从而减少了仿真所需的计算之间。

通过对试验数据的分析，我们可以看到，使用数据并行可以大幅度的提升计算效率。而使用改进的数据并行计算模型之后，相比之前可以提升大约3.45%的计算效率。这在大型 FDTD 仿真项目中会产生明显的区别。

表 3-4 传统数据并行方案和改进计算模型方案性能对比

函数	已用非独占时间（ms）		改进计算模型所节约时间
	传统数据并行方案	改进计算模型	
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] 仿真空间大小为1000×1000个Yee网格。

[2] 时间迭代次数为1000次。

表 3-5 改进计算模型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	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] 时间迭代次数为1000次。

表 3-6 改进计算模型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	7835.78	23247.73
compute	7819.76	23230.83
H_cmp	4.64	4.59
E_cmp	3.13	3.10

[1] 空间规模是2000×1000。

第4章 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;
    }
}
```

```
13
14 int main()
15 {
16     //declare for both Host and Device
17     int a[N], b[N], c[N];
18     int *dev_a, *dev_b, *dev_c;
19
20     //allocate memory for Device
21     cudaMalloc(&dev_a, N * sizeof(int));
22     cudaMalloc(&dev_b, N * sizeof(int));
23     cudaMalloc(&dev_c, N * sizeof(int));
24
25     //Initialize Host data
26     for (int i = 0; i < N; i++){
27         a[i] = i;
28         b[i] = i * i;
29     }
30
31     //Transfer Host data to Device.
32     cudaMemcpy(dev_a, a, N*sizeof(int),
          cudaMemcpyHostToDevice);
33     cudaMemcpy(dev_b, b, N*sizeof(int),
          cudaMemcpyHostToDevice);
34
35     //Excute Kernel
36     add <<<128, 128 >>>(dev_a, dev_b, dev_c);
37
38     //Transfer result from Device to Host
39     cudaMemcpy(c, dev_c, N*sizeof(int),
          cudaMemcpyDeviceToHost);
40
```

```
41      cudaFree(dev_a);
42      cudaFree(dev_b);
43      cudaFree(dev_c);
44
45      return 0;
46  }
```

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。

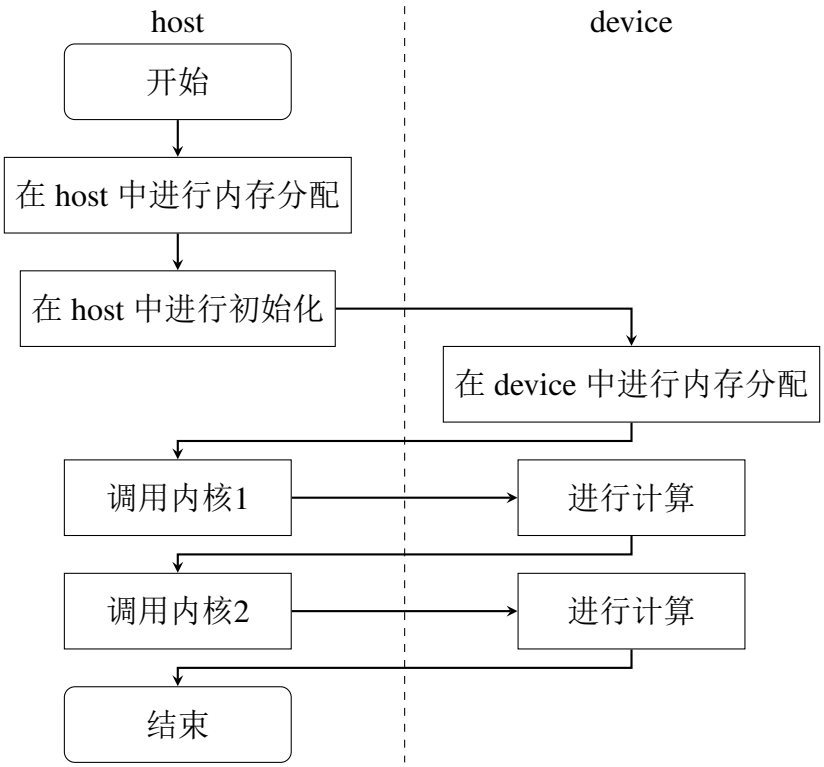

图 4-1 使用 CUDA 计算流程

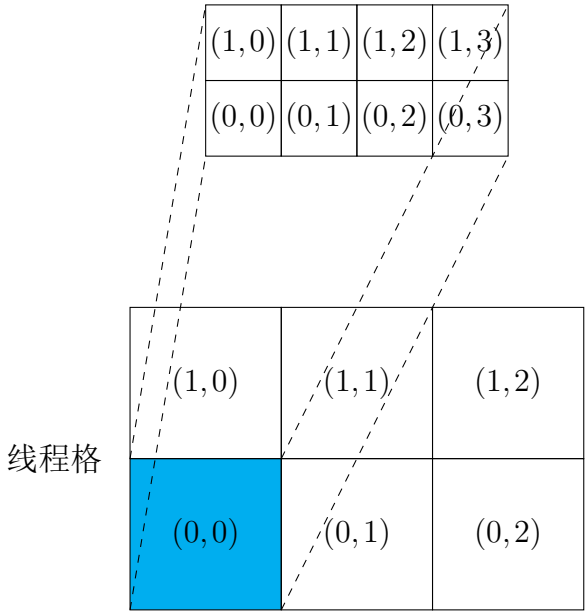

图 4-2 线程格和线程的二维层次结构示意图

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。

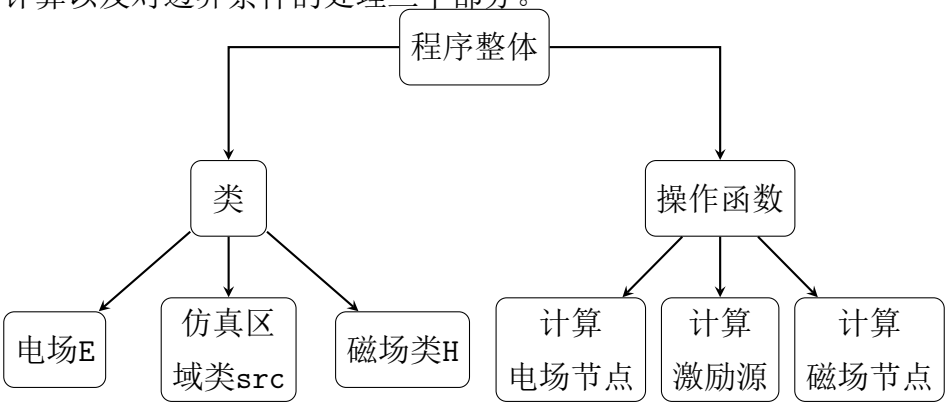

图 4-3 使用 CUDA 加速的程序的组成架构

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
```

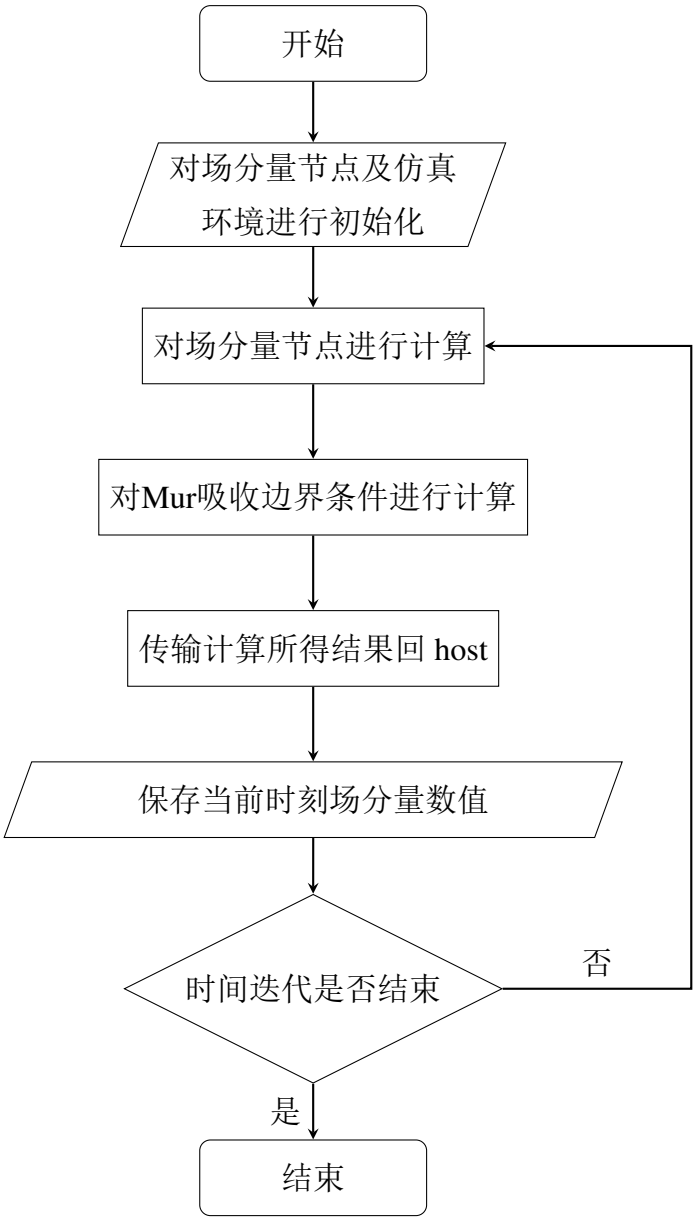

图 4-4 使用 CUDA 加速的 FDTD 程序流程图

```
17          //Hy(i,j)    -    Hy(i-1,j)
18          dif_Hy = Hy[y*ele_hy + x] - Hy[(y - 1)* ele_hy +
                x];
19          //Hx(i,j-1) -    Hx(i,j)
20          dif_Hx = Hx[y*ele_hx + (x - 1)] - Hx[y*ele_hx + x
                ];
21          Ez[y*ele_ex + x] += coe_Ez * (dif_Hx + dif_Hy);
22          tid += blockDim.x*gridDim.x;
23      }
24  }
```

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

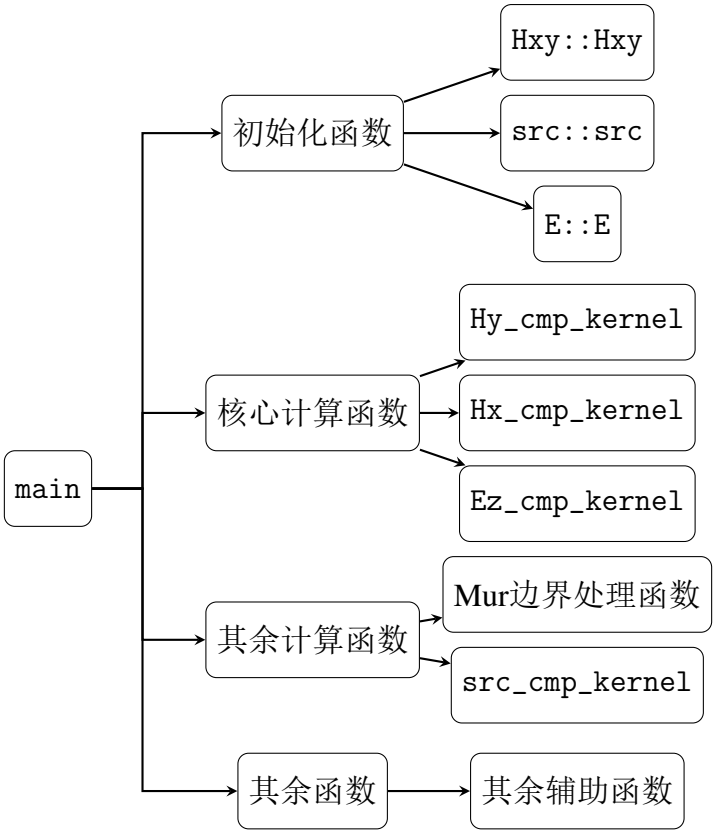

图 4-5 CUDA 加速的 FDTD 程序调用树

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 所节约时间
	传统串行方案	CUDA 加速方案	
$main$	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。

第5章 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。

参考文献

[1] K. Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media[J]. IEEE Transactions on Antennas and Propagation, 1966, 14(3):302–307

[2] C. Taylor, D.-H. Lam, T. Shumpert. Electromagnetic pulse scattering in time-varying inhomogeneous media[J]. IEEE Transactions on Antennas and Propagation, 1969, 17(5):585–589

[3] G. Mur. Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic-field equations[J]. Electromagnetic Compatibility, IEEE Transactions on, 1981(4):377–382

[4] J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves[J]. Journal of computational physics, 1994, 114(2):185–200

[5] Z. S. Sacks, D. M. Kingsland, R. Lee, et al. A perfectly matched anisotropic absorber for use as an absorbing boundary condition[J]. IEEE Transactions on Antennas and Propagation, 1995, 43(12):1460–1463

[6] S. D. Gedney. An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices[J]. IEEE Transactions on Antennas and Propagation, 1996, 44(12):1630–1639

[7] W. C. Chew, W. H. Weedon. A 3-D Perfectly Matched Medium from Modified Maxwell's Equations with Stretched Coordinates[J]. Microwave Opt. Tech. Lett, 1994, 7:599–604

[8] L. Zhang, W. Yu. Improving Parallel FDTD Method Performance Using SSE Instructions[J]. 2011:57–59

[9] M. Livesey, F. Costen, X. Yang. Double precision performance of streaming SIMD extensions instructions for the FDTD computation[J]. 2012:1–2

[10] Y. L. R. M. A. M. Wenhua Yu, Xiaoling Yang. Advanced FDTD Method[M]. ARTECH HOUSE, 2011

[11] S. E. Krakiwsky, L. E. Turner, M. M. Okoniewski. Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm[J]. 2004, 5:V–265–V–268 Vol.5

[12] J. A. Roden, S. D. Gedney. Convolution PML (CPML): An efficient FDTD implementation of the CFS–PML for arbitrary media[J]. Microwave and Optical Technology Letters, 2000, 27(5):334–339

[13] V. Demir. A simple GPU implementation of FDTD/PBC algorithm[J]. 2015:1–2

致 谢

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.

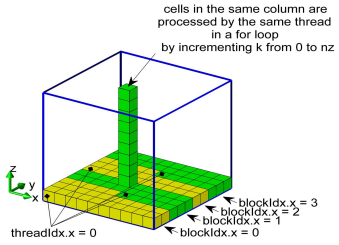

Figure 1. Mapping of threads to cells of an FDTD domain.

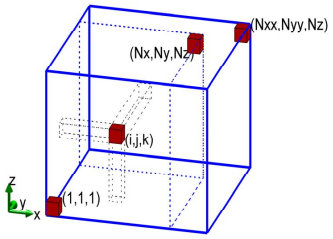

Figure 2. An FDTD problem space padded with additional cells.

```
__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.

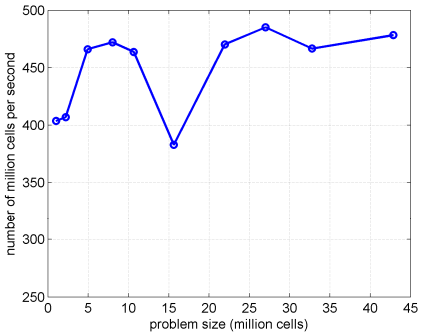

Figure 3. FDTD algorithm speed versus problem size.

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™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。