电磁场间断元 GPU 程序研究结题报告

指导老师：齐红星

小组成员：林泊润 唐瑜 邓雨君 方品 蒲飞

华东师范大学 物理系 上海 200241

摘　要: 利用 Maxwell 方程伽辽金间断有限元方法（DGM）本身具备的并行处理特性，在 GPU（Graphics Processing Unit）计算平台上，设计并实现基于 CUDA（Compute Unified Device Architecture）并行算法的程序，并以具体的金属谐振腔内的数值解与解析解比较来检验其可靠性。实验结果表明，程序计算结果完全正确，在性能良好的 GPU 硬件环境下，能实现不同程度的加速，并且保持计算精度不变。

关键字：伽辽金间断元；麦克斯韦方程；GPU；CUDA；并行计算

电磁仿真软件是电磁兼容设计和分析的重要工具。随着电子技术的进步，现代电子设备日益向集成化发展，系统的复杂程度也越来越高。传统的、针对芯片和简单子系统的仿真软件已经满足不了实际应用的要求，迫切需要能够解决系统级模拟的仿真软件。实现系统级电磁模拟，不仅要解决复杂形状和多尺寸同时存在的几何结构问题，而且要解决海量运算和数据存储问题。这是目前计算电磁学研究的重要方向。

近来发展的 Maxwell 方程伽辽金间断有限元方法（DGM）是当前实现复杂系统电磁计算最有潜力的方法。它不仅具有有限元方法的灵活建模能力，而且还有非常自然的网格加密和插值函数加密的特性。尤其对于瞬态计算，该算法还具备内在的并行处理特性。这为在高性能计算平台上实现系统级电磁仿真提供了可能。各种基于并行平台的 DGM 全波算法和软件研究是目前计算电磁学的热门研究课题。

目前国际上 DGM 的研究主要集中于协调网格和协调插值的 DGM 算法上，对非协调网格和非协调插值的研究相对较少。而非协调网格和插值对于 DGM 的适用性至关重要。非协调网格和非协调插值会对 DGM 算子特性产生重大影响。如何抑制非协调网格和非协调插值带来的算法不稳定和非物理伪解仍然是当前尚未解决的问题。我们前期工作在理论上建立了具有谱收敛特性的 DGM 时域算法，并在此基础上进一步提出了抑制算子伪解和局部时间步长积分的新技巧。

本研究的首要目的是编码实现这些算法，并用数值试验检验理论算法的可靠性。鉴于

GPU 是目前主流大型计算平台的架构体系，其运算速度比通用 CPU 处理器高两个数量级，具有非常强大的并行计算能力，本研究的另一个目的是建立基于 GPU 并行计算平台的 DGM 程序，为算法软件化和实用化进行探索。

一．有限元

（一）．简述

有限元分析是用较简单的问题代替复杂问题后再求解。它将求解域看成是由许多称为有限元的小的互连子域 g 组成，对每一单元假定一个合适的(较简单的）近似解，然后推导求解这个域总的满足条件(如结构的平衡条件），从而得到问题的解。这个解不是准确解，而是近似解，因为实际问题被较简单的问题所代替。由于大多数实际问题难以得到准确解，而有限元不仅计算精度高，而且能适应各种复杂形状，因而成为行之有效的工程分析手段。

有限元求解问题的基本步骤通常为：

第一步：问题及求解域定义：根据实际问题近似确定求解域的物理性质和几何区域。

第二步：求解域离散化：将求解域近似为具有不同有限大小和形状且彼此相连的有限个单元组成的离散域，习惯上称为有限元网络划分。显然单元越小（网格越细）则离散域的近似程度越好，计算结果也越精确，但计算量及误差都将增大，因此求解域的离散化是有限元法的核心技术之一。

第三步：确定状态变量及控制方法：一个具体的物理问题通常可以用一组包含问题状态变量边界条件的微分方程式表示，为适合有限元求解，通常将微分方程化为等价的泛函形式。

第四步：单元推导：对单元构造一个适合的近似解，即推导有限单元的列式，其中包括选择合理的单元坐标系，建立单元试函数，以某种方法给出单元各状态变量的离散关系，从而形成单元矩阵。

第五步：总装求解：将单元总装形成离散域的总矩阵方程（联合方程组），反映对近似求解域的离散域的要求，即单元函数的连续性要满足一定的连续条件。总装是在相邻单元结点进行，状态变量及其导数（可能的话）连续性建立在结点处。

第六步：联立方程组求解和结果解释：有限元法最终导致联立方程组。联立方程组的求解可用直接法、迭代法和随机法。求解结果是单元结点处状态变量的近似值。对于计算结果的质量，将通过与设计准则提供的允许值比较来评价并确定是否需要重复计算。

下面，我们分别以一维，二维的电磁学微分方程问题来阐述我们前期对有限元方法的认识。

（二）.有限元一维情况举例

用下列波动方程定义边值问题

$$\frac{d^2\emptyset}{dx^2}+2\emptyset=0 \qquad 0<x<1$$

其边界条件为$\emptyset|_{x=0}=0$和$\emptyset|_{x=1}=1$

用有限元方法求解，其求解过程:

1. 理论分析:

由一般式：$-\frac{d}{dx}\left(\alpha\frac{d\emptyset}{dx}\right)+\beta\emptyset=f$可知：

其中：$\begin{cases}\alpha=-1\\ \beta=2\\ f=0\end{cases}$

令在 L=1 上平均取 n=100 个单元

则 $l^e=0.01$

在各单元上：$\begin{cases}\alpha^e=-1\\ \beta^e=2\\ f^e=0\end{cases}$

则$K_{11}^e=K_{22}^e=\frac{\alpha^e}{l^e}+\beta^e\frac{l^e}{3}=-\frac{1}{0.01}+\frac{2*0.01}{3}=99.993$

$$K_{12}^e=K_{21}^e=\frac{\alpha^e}{l^e}+\beta^e\frac{l^e}{6}=-\frac{1}{0.01}+\frac{0.01}{6}=100.0033$$

$$\bar{K}^e=\begin{bmatrix} K_{11}^1 & K_{12}^1 & & & & \\ K_{21}^1 & K_{22}^1+K_{11}^2 & K_{12}^2 & & & \\ & K_{21}^2 & K_{22}^2+K_{11}^3 & K_{12}^3 & & \\ & & K_{21}^3 & K_{22}^3+K_{11}^4 & & \\ & & \cdots\cdots\cdots & & & \\ & & K_{21}^{99} & K_{22}^{99}+K_{11}^{100} & K_{12}^{100} & \\ & & & K_{21}^{100} & K_{22}^{100} & \end{bmatrix}$$

$[b^e]=0$

强加边界点代值得

$$\begin{bmatrix} 1 & 0 & . & . & . \\ -199.987 & 100.0033 & . & . & . \\ . & 100.0033 & -199.987 & 100.0033 & . \\ . & . & 100.0033 & -199.987 & 100.0033 \\ . & . & \cdots\cdots & . & . \\ . & . & 100.0033 & -199.987 & 100.0033 \\ 0 & . & . & 100.0033 & -199.987 \\ 0 & 0 & . & . & 1 \end{bmatrix}\begin{bmatrix}\emptyset_1\\ \emptyset_2\\ \emptyset_3\\ .\\ .\\ .\\ \emptyset_{100}\\ \emptyset_{101}\end{bmatrix}=\begin{bmatrix}0\\0\\0\\.\\.\\.\\0\\1\end{bmatrix}$$

2．通过程序编写，运用高斯消元法即可求解出单元结点上电势值。

3．通过作图分析，将微分方程精确解与通过有限元方法求解结果所得曲线进行对比，见图 1。

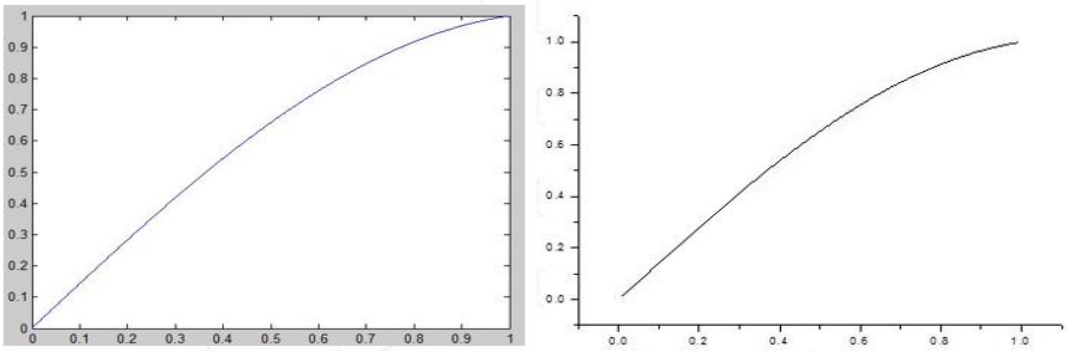


图 1. 微分方程精确解与有限元方法求解结果曲线图对比

结果分析：通过有限元方法求解一维微分方程，在所在区域离散化的单元足够大时，其结果与精确解曲线能保持几乎一致。

（三）. 有限元二维情况举例

考虑如下图 2 所示的矩形区域。假设：在 AB 上，$\emptyset = 1$：在 CD 上，$\emptyset = 0$.进一步假设：在 AC 和 BD 两条边上，满足齐次诺曼边界条件。根据下面列出的，解泊松方程：

$$\frac{\partial^2 \emptyset}{\partial x^2} + \frac{\partial^2 \emptyset}{\partial y^2} + 1 = 0$$

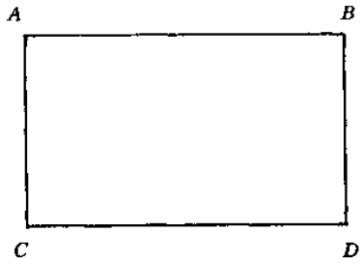


图 2

1. 理论求解过程：

   按下图 3 在矩形框中划分出 162 个三角形单元格，共 100 个结点,并标注坐标,令方形格上方的三角形单元为奇单元，方形格下方的三角形单元为偶单元。且令每个方形格面积为 1。

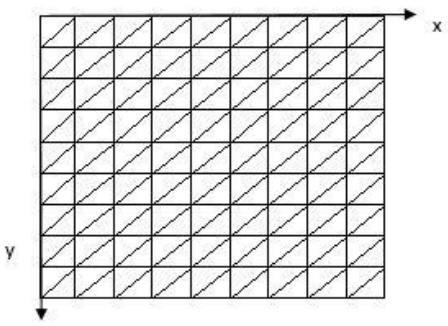


图 3

则每个三角形面积：

$$\Delta^e = 0.5$$

如此：

$$b_i^e = \frac{\Delta^e}{3} f^e = -\frac{1}{6}$$

$$k_{ij}^e = \frac{1}{\Delta^e}\left(\alpha_x^e b_i^e b_j^e + \alpha_y^e c_i^e c_j^e\right) + \frac{\Delta^e}{12}\beta^e(1+\delta_{ij})$$

对于微分方程：

$$\frac{\partial^2 \emptyset}{\partial x^2} + \frac{\partial^2 \emptyset}{\partial y^2} + 1 = 0$$

由一般式

$$-\frac{\partial}{\partial x}\left(\alpha_x \frac{\partial}{\partial x}\right) - \frac{\partial}{\partial y}\left(\alpha_y \frac{\partial \emptyset}{\partial y}\right) + \beta\emptyset = f$$

可知 $\begin{cases} \alpha_x = -1 \\ \alpha_y = -1 \\ \beta = 0 \\ f = -1 \end{cases}$

则$k_{ij}^e = -\frac{1}{2}\left(b_i^e b_j^e + c_i^e c_j^e\right)$

$$= -\frac{1}{2}\left[\left(y_{i+1}^e - y_{i+2}^e\right)\left(y_{j+1}^e - y_{j+2}^e\right) + \left(x_{i+2}^e - y_{i+1}^e\right)\left(x_{j+2}^e - y_{j+1}^e\right)\right]$$

通过计算可以知道：$k_{11}^e = k_{33}^e = -\frac{1}{2}$

$$k_{12}^e = k_{21}^e = k_{23}^e = k_{32}^e = \frac{1}{2}$$

$$k_{13}^e = k_{31}^e = 0$$

则对于奇单元：

$$k_{11*11}^{2n+1} = \begin{pmatrix} -1 & \frac{1}{2} & \cdots & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{2} & 0 & \cdots & -2 \end{pmatrix}$$

对于偶单元：

$$k_{12*12}^{2n}=\begin{pmatrix}0 & 0 & \cdots & 0 & 0\\ 0 & -\frac{1}{2} & \cdots & 0 & \frac{1}{2}\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ 0 & 0 & \cdots & -\frac{1}{2} & \frac{1}{2}\\ 0 & \frac{1}{2} & \cdots & \frac{1}{2} & -1\end{pmatrix}$$

对$k^1+k^2+k^3+\cdots+k^{161}+k^{162}$求和即可求得矩阵$[k]_{100*100}$.

因为在边界 AB 上，$\emptyset=1$：在 CD 上，$\emptyset=0$，应用边界条件可得：

$$k_{1,1}=k_{11,11}=k_{21,21}=\cdots=k_{81,81}=k_{91,91}=1$$

$$k_{10,10}=k_{20,20}=\cdots=k_{90,90}=k_{100,100}=1$$

当 i≠ $10n$时，$k_{10n,i}=0.$ (其中$n=1,2\ldots10;i=1,2,\ldots,100$)

当 i≠ $10n+1$ 时，$k_{10n+1,i}=0.$ (其中$n=0,1,\ldots,9;i=1,2,,\ldots,100$)

对于矩阵[b]，由$b_i^e \Rightarrow b_{n(i,e)}$

以b^1,b^2为例：

$$b_{12*1}^1=\begin{pmatrix}b_3^1\\ b_1^1\\ 0\\ \vdots\\ 0\\ b_2^1\\ 0\end{pmatrix}\qquad b_{12*1}^2=\begin{pmatrix}0\\ b_1^2\\ 0\\ \vdots\\ 0\\ b_3^2\\ b_2^2\end{pmatrix}\qquad b_{12*1}^1+b_{12*1}^2=\begin{pmatrix}b_3^1\\ b_1^1+b_1^2\\ 0\\ \vdots\\ 0\\ b_2^1+b_3^2\\ b_2^2\end{pmatrix}$$

同理：

$$b_{13*1}^3+b_{13*1}^4=\begin{pmatrix}0\\ b_3^1\\ b_1^1+b_1^2\\ 0\\ \vdots\\ 0\\ b_2^1+b_3^2\\ b_2^2\end{pmatrix}$$

即$b_{13*1}^3+b_{13*1}^4$为$b_{12*1}^1+b_{12*1}^2$的各元素值不变，其位置各向下平移一行，

$b_{14*1}^5+b_{14*1}^6$，$b_{15*1}^7+b_{15*1}^8\ldots\ldots$依次类推。直到第$b_{22*1}^{19}+b_{22*1}^{20}$相对于$b_{20*1}^{17}+b_{20*1}^{18}$向下平移 2 行。$b_{23*1}^{21}+b_{23*1}^{22}$相对于$b_{22*1}^{19}+b_{22*1}^{20}$向下平移一行。

直到$b_{32*1}^{37}+b_{32*1}^{38}$相对于$b_{30*1}^{35}+b_{30*1}^{36}$向下平移 2 行。

依次类推，以向下 8 次一行，1 次两行为单位，重复 9 次，最后 1 次下移 2 行结束。

即可完成矩阵[b]的合成。

现在已求得矩阵[k]和[b],可以建立矩阵方程式：$[k]_{100*100}[\emptyset]_{100*1}=[b]_{100*1}$

2. 根据上面理论分析，用计算机编写程序，计算出 100 个结点上$[\emptyset]_{100*1}$的结果值。
3. 运用 tecplot 软件，绘制出微分方程的二维有限元图像。
4. 重新对上微分方程，对区域划分 30*30 结点，40*40 结点和 50*50 结点，根据 1—3 的方法和步骤，运用 tecplot 软件，绘制含更多单元的图像。
5. 其图像结果可参看图 4。

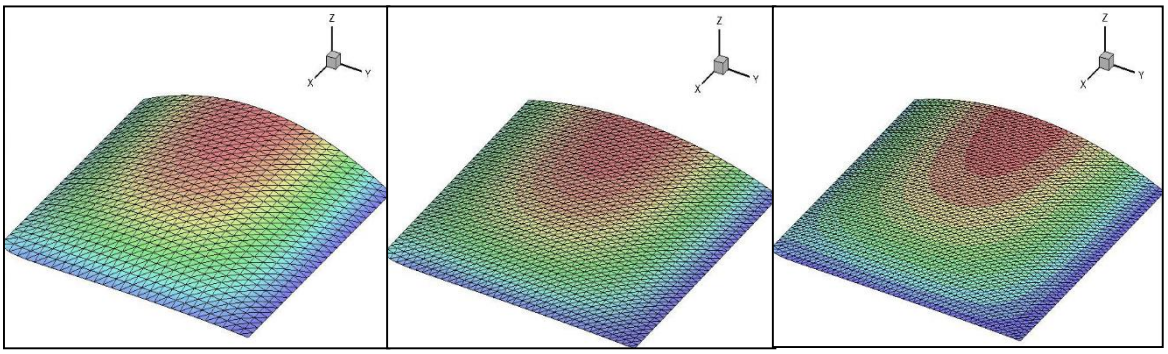


图 4.不同结点数所反映的电磁场电势图

6. 结果总结：

1）.图像对电磁场的电势显示与现实结果非常符合，不会因为插入点的多少而改变电磁场电势的分布情况；

2）.当在空间中插入的结点数越多，越能与现实的真实电势相符。但在获得较高精度的同时，对计算机的计算量也提高了更高的要求。

二． 间断有限元简介及与并行的计算的相关关系

间断有限元方法是 1973 年由 Reed 和 Hill 首先提出，并应用于救解中子输运方程。但这种方法长期以来一直没有得很好的研究和应用。直到 20 世纪 80 年代后期和 90 年代。Cockbum 和 Shu 等结合 Runge-Kutta 方法，将间断有限元方法推广到非线性一维守恒方程和和方程组，高维守恒方程和方程组，并给出了部分收敛性理论证明后，这一方法才引起人们的注意，并逐渐开始应用于电磁场计算领域。

间断有限元方法应用于电磁场计算领域，并受到人们的关注，主要因其保持了通常有限元方法的优点：

能够处理复杂的区域边界和具有复杂的边界条件问题，并获得与区域内部一致的计算精度；易于网格加密和高精度处理边界条件，实现自适应计算；

可以得到任意阶精度的格式。同时又具有很好的局部紧致性，构造的高阶格式不需要非常宽的模板；

同时由于于间断有限元方法吸收了差分方法的一些特点，能够显示求解。因此容易实现并行计算法

具有很好的稳定性，满足l^2稳定性和熵相容性。

间断有限元方法的缺点是程序设计比较复杂和计算量大。然而近十年来为，随着计算机条件不断的改善，各种大型向量计算机和并行计算机的相继问世，间断有限元方法能够较容易地把求解二维问题的方法推广到三维问题。实现自适应算法和并行算法。

本课题的研究就基于 GPU 强大的并行计算能力处理电磁场间断有限元在计算复杂和数数据量大的问题上展开研究

三．GPU 程序编写

（一）．概述

本程序使用的语言是 CUDA C，完成的功能是金属谐振腔内的数值解与解析解比较。输入由 ELIST.BIN、NLIST.BIN 等文件提供，输出为 EvsT_E4H3PE.DAT 文件。

本程序的主要计算集中在 FieldJump()函数和 UPEM()函数上。

（二）．程序的执行过程

在进入到真正的 GPU 计算之前，需要做一些准备工作，包括初始化，读入数据等，而这一部分是由 CPU 来完成的。具体的执行如图 5 所示，这是程序一开始所需要做的工作，包括读计算空间网格和顶点数据、计算网格关联矩阵、计算物理单元和参考单元间变换的雅可比矩阵、读参考单元上的数据、基本参数赋值以及建立计算单元等。

```
int main()
{
    #define _FILE_FORMAT_CONVERT_
    #ifdef  _FILE_FORMAT_CONVERT_

    int nVertTot, nElemTot;

    nVertTot = NodesListFmtToDataFmt( );
    nElemTot = ElemsListFmtToDataFmt( );
    NodesDataFmtToBinFmt( nVertTot );
    ElemsDataFmtToBinFmt( nElemTot );

    #endif

    // 读计算空间网格和顶点数据
    ReadElem( );
    ReadVert( );

    ElemIncMat( );       // 计算网格关联矩阵
    MatJacobi( );        // 计算物理单元和参考单元间变换的雅可比矩阵
    ReadRefElemInfo( ); // 读参考单元上的数据
    InitGlobPara( );     // 基本参数赋值

    InitElem( );         // 建立计算单元
```

图 5.计算开始前的准备工作

当 CPU 上的工作完成后，就可以进入到 GPU 的工作上了。为了使程序的结构更加清晰，CUDA C 的代码被集中在一个函数 NDGTD3D_RKwithCuda()中。其中函数执行的成功与否，会返回给变量 cudaStatus，其类型是 cudaError_t，可以理解该类型是专门用于表示 CUDA 函数

的执行状态。因此，当函数执行结束后，将会检测 cudaStatus 的值，如果表示不成功，将会标准错误输出。

```
    // NDGTD3D_RK in parallel.
    cudaError_t cudaStatus = NDGTD3D_RKwithCuda();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "NDGTD3D_RKwithCuda failed!");
        return 1;
    }

    /* ... */
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    getchar();

    return 0;
}
```

图 6. 函数 NDGTD3D_RKwithCuda()

以上就是程序的基本框架，也是 main()函数中的主要内容。而程序真正的计算在函数 NDGTD3D_RKwithCuda()中，因此接下来将剖析 NDGTD3D_RKwithCuda()函数。

在 NDGTD3D_RKwithCuda()函数中，首先需要声明变量，以及对一些变量进行初始化。

```
// Helper function for using CUDA to NDGTD3D_RK in parallel.
cudaError_t NDGTD3D_RKwithCuda()
{
    const int length = 1<<15;
    double t = 0.;
    clock_t uStart, uFinish;
    uStart = clock();

    ofstream EvsT( "EvsT_E4H3PE.DAT", ios::out );

    // 错误检测
    int flag = 0 , *dev_flag;
    // Host端data , Device端data
    ELMT  *gpu_EL , *cpu_EL;

    int *gpu_IE2D , *gpu_IH2D;
    FLT *gpu_RK4a , *gpu_RK4b;
    FLT *gpu_EE , *gpu_EH , *gpu_HE , *gpu_HH;
    FLT *gpu_IVME , *gpu_IVMH;
    size_t d_pitchBytes;                    // 字节对齐
    int Ht = 4;                  // 高度
    int Wd_IE2D = NE2D;                     // 宽度
    int Wd_IH2D = NH2D;                     // 宽度

    dim3 blocks(NELEM, 4);
    dim3 blocks2(NELEM, 6);
    dim3 threads(NE2D, NH2D);
    dim3 threads2_1(NE3D, NH3D);
    dim3 threads2_2(8, 8, 8);
    cudaError_t cudaStatus;
```

图 7. 变量声明以及初始化

接着，需要选择 GPU 设备。如果在多 GPU 系统下，一般需要选择其中一个 GPU 作为 CUDA 运行的硬件环境。如果是在单 GPU 系统下，则选择第 0 个 GPU 硬件设备即可。

```
// Choose which GPU to run on, change this on a multi-GPU system.
cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
    goto Error;
}
```

图 8. 选择 GPU 设备

然后，需要分配 GPU 内存以及 CPU 内存。由于 CUDA 是运行在 GPU 上的，而 CPU 上的内存是无法在 GPU 中读取的，GPU 有属于自己的内存。因此在 GPU 中进行计算的数据要存在 GPU 内存才能被 GPU 读写。也即是说，要将数据放在 GPU 上计算，需要先为这些数据分配 GPU 内存，再将这些数据从 CPU 内存拷贝到 GPU 内存上。

其中，在 GPU 中分配一维内存使用函数 cudaMalloc ，其函数原型为：

cudaError_t cudaMalloc(void **devPtr, size_t size)

分配二维内存则可使用函数 cudaMallocPitch ，其函数原型为：

cudaError_t cudaMallocPitch(void **devPtr, size_t *pitch, size_t width, size_t height)

```
// Allocate GPU buffers for three vectors (two input, one output)
cudaStatus = cudaMalloc((void**)&gpu_EL, NELEM * sizeof(ELMT));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}
```

图 9. 分配一维 GPU 内存

```
cudaStatus = cudaMallocPitch((void**) &gpu_IE2D, &d_pitchBytes, sizeof(int) * Wd_IE2D, Ht);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMallocPitch failed!");
    goto Error;
}
```

图 10. 分配二维 GPU 内存

在本程序中，需要为其分配一维内存的数据有 EL、RK4a、RK4b，需要为其分配二维内存的数据则有 IE2D、IH2D、EE、EH、HE、HH、IVME、IVMH。为了区分 GPU 内存和 CPU 内存上的数据，将存储在 GPU 内存中的数据变量名加上前缀 gpu，如 EL 和 gpu_EL。

除了分配 GPU 内存外，还需分配 CPU 内存。因为当数据在 GPU 内存中被处理后，需要写回到磁盘中，而 GPU 内存的数据是不能直接写到磁盘中的，因此需要先把数据从 GPU 内存传回 CPU 内存，再由 CPU 把数据写到磁盘中。

```
// Allocate CPU buffers
cpu_EL = (ELMT*)calloc(NELEM, sizeof(ELMT));
```

图 11. 分配 CPU 内存

分配完内存后，就需要把数据从 CPU 内存拷贝到 GPU 内存。所使用的函数是 cudaMemcpy ，其函数原型为：

cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)

需要说明的是，参数 cudaMemcpyKind 表示数据传输的方向，当 cudaMemcpyKind 的值为 cudaMemcpyHostToDevice 时，表示数据从 CPU 内存传送到 GPU 内存，当 cudaMemcpyKind 的值为 cudaMemcpyDeviceToHost 时，则表示数据从 GPU 内存传送到 CPU 内存。

拷贝一维数据到 GPU 内存中，可直接使用 cudaMemcpy 函数。

```
// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(gpu_EL, &EL[0], NELEM * sizeof(ELMT), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}
```

图 12. 将一维数据从 CPU 内存拷贝到 GPU 内存中

而拷贝二维数据到 GPU 内存中的话，可通过 for 语句多次调用 cudaMemcpy 函数来实现。

```
for(int row = 0; row < Ht; ++row) {
    cudaStatus = cudaMemcpy(&gpu_IE2D[row*(d_pitchBytes/sizeof(int))], &IE2D[row][0], sizeof(int)*Wd_IE2D, cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
}
```

图 13. 将二维数据从 CPU 内存拷贝到 GPU 内存中

以上所说的 GPU 内存是全局内存，这也是 CUDA 环境下最常 22 用的内存类型。而 CUDA 除了有全局内存外，还有纹理内存、常量内存、共享内存等类型。而其他内存有自己的申请方式。

比如纹理内存就需要额外的函数对存 33 储在 GPU 内存中的数据进行绑定，绑定所调用的函数为 cudaBindTexture2D，其函数原型为：

cudaError_t cudaBindTexture2D(size_t *offset, const struct textureReference *texref, const void *devPtr, const struct cudaChannelFormatDesc *desc, size_t width, size_t height, size_t pitch);

```
// 绑定纹理内存
cudaStatus = cudaBindTexture2D_IE2D(gpu_IE2D, Wd_IE2D, Ht, d_pitchBytes );
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaBindTexture2D failed!");
    goto Error;
}
```

图 14-1. 绑定纹理内存

```
cudaError_t cudaBindTexture2D_IE2D(const void *devPtr, size_t width, size_t height, size_t pitch)
{
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<int>();
    return cudaBindTexture2D( NULL, texIE2D, devPtr, desc, width, height, pitch );
}
```

图 14-2. 绑定纹理内存 续

而常量内存则不能使用 cudaMemcpy 函数来传输数据，需要用另外一个函数 cudaMemcpyToSymbol，其函数原型为：

cudaError_t cudaMemcpyToSymbol(const void *symbol, const void *src, size_t count, size_t offset __dv(0), enum cudaMemcpyKind kind __dv(cudaMemcpyHostToDevice))

```
// 将值复制到常量内存中
cudaStatus = cudaMemcpyToSymbol_NELEM();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpyToSymbol failed!");
    goto Error;
}
```

图 15-1. 将数据从 CPU 内存拷贝到 GPU 常量内存中

```
cudaError_t cudaMemcpyToSymbol_NELEM()
{
    // 注意一点，第一个参数无论是数组还是变量，都不需要使用&符号
    return cudaMemcpyToSymbol( dev_NELEM, &NELEM, sizeof(int));
}
```

图 15-2.将数据从 CPU 内存拷贝到 GPU 常量内存中 续

当数据准备完成后，就真正进入到计算部分了。为清晰地了解计算部分的代码结构，可参考图 16 的 CPU 版本下计算部分的简化代码。

```
for( int its = 0; its < 1<<15; ++ its )
{
    for( int iStage = 0; iStage < 5; ++ iStage )
    {
        FieldJump( );

        UPEM( iStage, t + RK4c[ iStage ] * DT );
    }
}
```

图 16. CPU 版本下计算部分的简化代码

计算部分的代码结构很容易理解，即是把 FieldJump 函数和 UPEM 函数执行$(1 << 15) \times 5$遍。而 GPU 版本下的结构也是一样的，只是在 CPU 版本的基础上多了一些检错函数和同步函数。

当计算部分成功结束后，并将数据进行输出，程序就执行完毕。

以上就是程序的执行过程。

（三）. FieldJump 函数

FieldJump 函数所要做的任务是计算四个表面上的场量跃变。

CPU 版本的 FieldJump 的函数原型为：

```
void FieldJump( void );
```

在程序中的调用方式为：

```
FieldJump( );
```

而 GPU 版本的 FieldJump 的函数原型为：

```
__global__ void FieldJump( ELMT *gpu_EL, int *gpu_flag );
```

在程序中的调用方式为：

```
FieldJump<<<blocks, threads>>>( gpu_EL, dev_flag );
```

```
// Launch a kernel on the GPU.
FieldJump<<<blocks, threads>>>( gpu_EL, dev_flag );

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "FieldJump launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching FieldJump!\n", cudaStatus);
    goto Error;
}
```

图 17. GPU 版本下 FieldJump 函数的调用

其中传递的参数 gpu_EL 和 dev_flag 均为全局内存的变量。而函数中的尖括号选项<<<blocks, threads>>>则指明了分配给 FieldJump 函数的线程块数量以及每个线程块所拥有的线程数量。另外，FieldJump 函数后面的 2 个函数 cudaGetLastError()和 cudaDeviceSynchronize()的作用分别是用于检测错误和同步。

```
// 计算四个表面上的场量跃变
__global__ void FieldJump( ELMT *gpu_EL, int *gpu_flag )
{
    int iElemNB , iFaceNB ;
    int iElem = blockIdx.x , iFace = blockIdx.y ;
    int iNodeE = threadIdx.x , iNodeH = threadIdx.y ;

    int temp1 , temp2;

    if( iElem < dev_NELEM && iFace < 4 )
    {
        iElemNB = gpu_EL[ iElem ].ElemNB[ iFace ];
        iFaceNB = gpu_EL[ iElem ].FaceNB[ iFace ];

        if( iElemNB != -1 && iFaceNB != -1) // 外部场减去内部场
        {
            if( iNodeE < dev_NE2D && iNodeH == 0 )
            {
                temp1 = tex2D(texIE2D, iNodeE, iFaceNB);
                temp2 = tex2D(texIE2D, iNodeE, iFace);
                gpu_EL[ iElem ].dEx[ iFace ][ iNodeE ] = gpu_EL[ iElemNB ].EX[ temp1 ]
                                                       - gpu_EL[ iElem   ].EX[ temp2 ];
                gpu_EL[ iElem ].dEy[ iFace ][ iNodeE ] = gpu_EL[ iElemNB ].EY[ temp1 ]
                                                       - gpu_EL[ iElem   ].EY[ temp2 ];
                gpu_EL[ iElem ].dEz[ iFace ][ iNodeE ] = gpu_EL[ iElemNB ].EZ[ temp1 ]
                                                       - gpu_EL[ iElem   ].EZ[ temp2 ];
            }
```

图(a) GPU 版本的 FieldJump 函数部分代码

```
// 计算四个表面上的场量跃变
void FieldJump( void )
{
    int iElem, iFace, iNodeE, iNodeH, iElemNB, iFaceNB;

    #pragma omp parallel for default( shared )\
    private( iElem, iFace, iNodeE, iNodeH, iElemNB, iFaceNB )
    for( iElem = 0; iElem < NELEM; ++ iElem )
    for( iFace = 0; iFace < 4    ; ++ iFace )
    {
        iElemNB = EL[ iElem ].ElemNB[ iFace ];
        iFaceNB = EL[ iElem ].FaceNB[ iFace ];

        if( iElemNB != -1 && iFaceNB != -1) // 外部场减去内部场
        {
            for( iNodeE = 0; iNodeE < NE2D; ++ iNodeE )
            {
                EL[ iElem ].dEx[ iFace ][ iNodeE ] = EL[ iElemNB ].EX[ IE2D[ iFaceNB ][ iNodeE ] ]
                                                   - EL[ iElem   ].EX[ IE2D[ iFace   ][ iNodeE ] ];
                EL[ iElem ].dEy[ iFace ][ iNodeE ] = EL[ iElemNB ].EY[ IE2D[ iFaceNB ][ iNodeE ] ]
                                                   - EL[ iElem   ].EY[ IE2D[ iFace   ][ iNodeE ] ];
                EL[ iElem ].dEz[ iFace ][ iNodeE ] = EL[ iElemNB ].EZ[ IE2D[ iFaceNB ][ iNodeE ] ]
                                                   - EL[ iElem   ].EZ[ IE2D[ iFace   ][ iNodeE ] ];
            }
```

图(b) CPU 版本的 FieldJump 函数部分代码

图 18. GPU 版本和 CPU 版本的 FieldJump 函数部分代码对比

在 FieldJump 函数中，被写的数据是 gpu_EL，而只读的数据有 texIE2D、texIH2D，因此可将 texIE2D 和 texIH2D 存储为纹理内存。纹理内存的特点是数据只允许读，而且当访问的数据具有空间局部性的时候，读取速度会加快。另外，一些只读的变量（如 dev_NELEM、dev_NE2D、dev_NH2D 等）则可以存储在常量内存中，也可以减少访问数据时的开销。常量内存的特点是当某一部分的所有线程都访问相同的只读数据时，可以减少内存带宽，加快读取速度。

（四）. UPEM 函数

UPEM 函数所做的计算是更新电磁场。

在 CPU 版本中，UPEM 函数的函数原型为：

```
void UPEM( const int &id, const FLT &t );
```

在程序中的调用方式为：

```
UPEM( iStage, t + RK4c[ iStage ] * DT );
```

而在 GPU 版本中，UPEM 函数被做了大幅度的修改。UPEM 函数被拆分成了许多小的函数，拆分的原则是将 UPEM 函数中不具有数据相关性的代码尽可能地分开，并让这些函数独立执行，以提高执行速度。

首先，根据 UPEM 函数的代码结构，可将 UPEM 分为四大部分。其中，第一部分为处理微分矩阵部分，将其独立出来并命名为 UPEM_1；而剩下的部分则是通量部分，其中第二部分为通量部分中的计算（质量逆矩阵与通量乘积的累加）部分，将其独立出来并命名为 UPEM_2；第三部分是通量部分中的惩罚项处理部分，将其独立出来并命名为 UPEM_3；第四部分则为通量部分中的累加计算结果部分，将其独立出来并命名为 UPEM_4。

除此以外，每一个大部分中又分为电场部分和磁场部分。而电场部分和磁场部分又各自可被分成 3 个小部分，这 3 个部分分别为 X 部分、Y 部分和 Z 部分。其中：

电场的 X 部分被命名为 UPEM_N_0；

电场的 Y 部分被命名为 UPEM_N_1；

电场的 Z 部分被命名为 UPEM_N_2；
磁场的 X 部分被命名为 UPEM_N_3；
磁场的 Y 部分被命名为 UPEM_N_4；
磁场的 Z 部分被命名为 UPEM_N_5。

命名中的 N 表示四大部分中的某一个，如第一部分中的电场 X 部分则是 UPEM_1_0，以此类推。

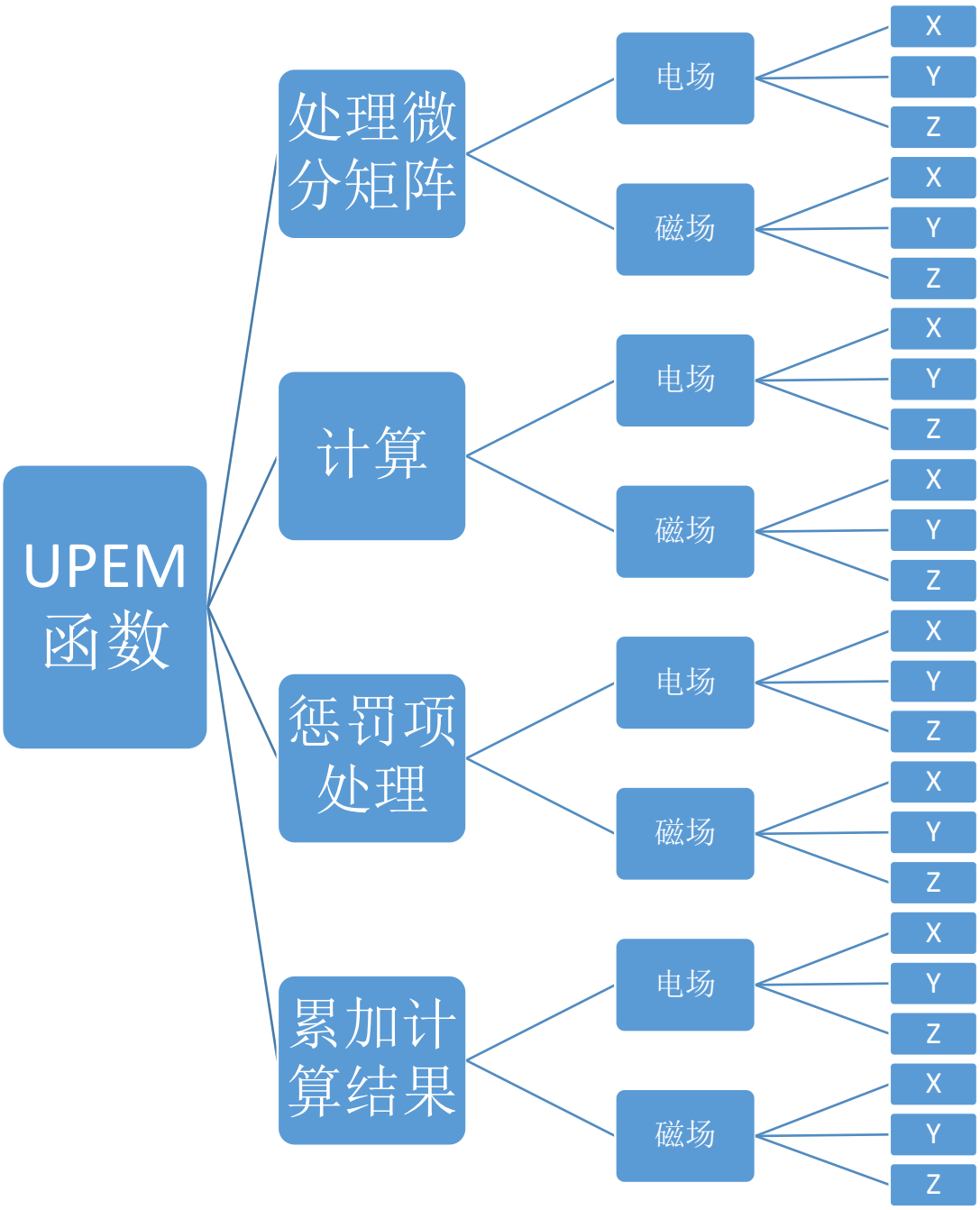


图 19. UPEM 函数的结构

由 UPEM 函数的结构可知，UPEM 函数将被分成 24 个函数，这 24 个函数的函数原型为：

```
__global__ void UPEM_1_0( ELMT *EL );
__global__ void UPEM_1_1( ELMT *EL );
```

```
__global__ void UPEM_1_2( ELMT *EL );

__global__ void UPEM_1_3( ELMT *EL );
__global__ void UPEM_1_4( ELMT *EL );
__global__ void UPEM_1_5( ELMT *EL );

__global__ void UPEM_2_0( ELMT *EL, FLT *EE, FLT *EH, FLT *IVME );
__global__ void UPEM_2_1( ELMT *EL, FLT *EE, FLT *EH, FLT *IVME );
__global__ void UPEM_2_2( ELMT *EL, FLT *EE, FLT *EH, FLT *IVME );

__global__ void UPEM_2_3( ELMT *EL, FLT *HE, FLT *HH, FLT *IVMH );
__global__ void UPEM_2_4( ELMT *EL, FLT *HE, FLT *HH, FLT *IVMH );
__global__ void UPEM_2_5( ELMT *EL, FLT *HE, FLT *HH, FLT *IVMH );

__global__ void UPEM_3_0( ELMT *EL, FLT *EE );
__global__ void UPEM_3_1( ELMT *EL, FLT *EE );
__global__ void UPEM_3_2( ELMT *EL, FLT *EE );

__global__ void UPEM_3_3( ELMT *EL, FLT *HH );
__global__ void UPEM_3_4( ELMT *EL, FLT *HH );
__global__ void UPEM_3_5( ELMT *EL, FLT *HH );

__global__ void UPEM_4_0( ELMT *EL );
__global__ void UPEM_4_1( ELMT *EL );
__global__ void UPEM_4_2( ELMT *EL );

__global__ void UPEM_4_3( ELMT *EL );
__global__ void UPEM_4_4( ELMT *EL );
__global__ void UPEM_4_5( ELMT *EL );
```

而这 24 个 UPEM 函数在程序中的调用为：

```
UPEM_1_0<<<NELEM, threads2_1>>>( gpu_EL );
UPEM_1_1<<<NELEM, threads2_1>>>( gpu_EL );
UPEM_1_2<<<NELEM, threads2_1>>>( gpu_EL );

UPEM_1_3<<<NELEM, threads2_1>>>( gpu_EL );
UPEM_1_4<<<NELEM, threads2_1>>>( gpu_EL );
UPEM_1_5<<<NELEM, threads2_1>>>( gpu_EL );

UPEM_2_0<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE, gpu_EH, gpu_IVME );
UPEM_2_1<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE, gpu_EH, gpu_IVME );
UPEM_2_2<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE, gpu_EH, gpu_IVME );

UPEM_2_3<<<NELEM, threads2_2>>>( gpu_EL, gpu_HE, gpu_HH, gpu_IVMH );
UPEM_2_4<<<NELEM, threads2_2>>>( gpu_EL, gpu_HE, gpu_HH, gpu_IVMH );
```

```
UPEM_2_5<<<NELEM, threads2_2>>>( gpu_EL, gpu_HE, gpu_HH, gpu_IVMH );

UPEM_3_0<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE );
UPEM_3_1<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE );
UPEM_3_2<<<NELEM, threads2_2>>>( gpu_EL, gpu_EE );

UPEM_3_3<<<NELEM, threads2_2>>>( gpu_EL, gpu_HH );
UPEM_3_4<<<NELEM, threads2_2>>>( gpu_EL, gpu_HH );
UPEM_3_5<<<NELEM, threads2_2>>>( gpu_EL, gpu_HH );

UPEM_4_0<<<NELEM, NE3D>>>( gpu_EL );
UPEM_4_1<<<NELEM, NE3D>>>( gpu_EL );
UPEM_4_2<<<NELEM, NE3D>>>( gpu_EL );

UPEM_4_3<<<NELEM, NH3D>>>( gpu_EL );
UPEM_4_4<<<NELEM, NH3D>>>( gpu_EL );
UPEM_4_5<<<NELEM, NH3D>>>( gpu_EL );
```

（五）. 总结

程序在执行过程中，会将部分计算的值输出到标准输出中，以便查看。

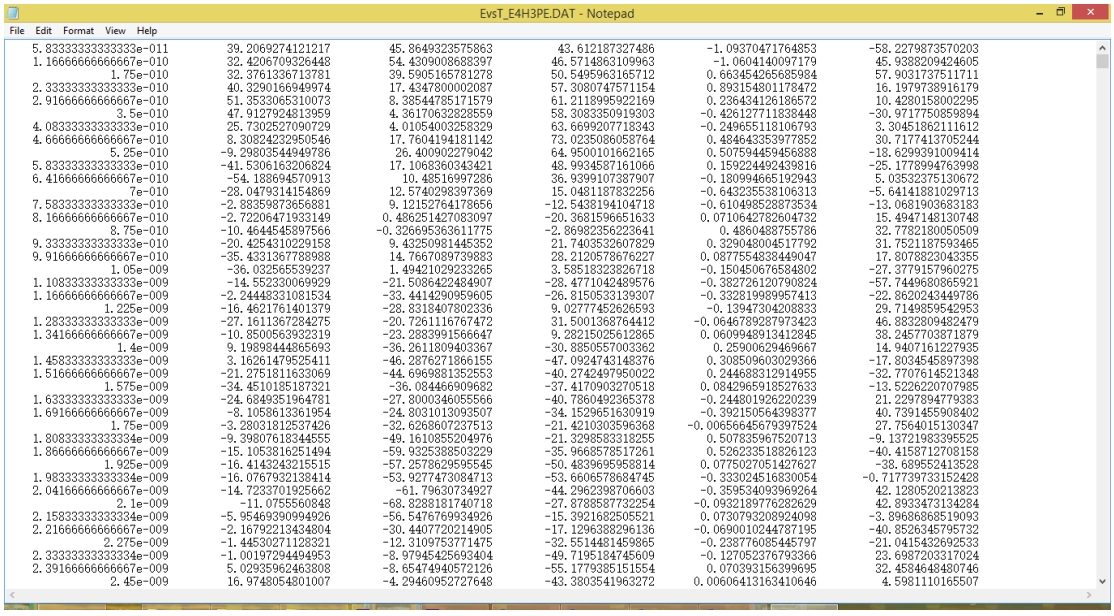
```
NodesListFmtToDataFmt ......
      100%       Total Nodes:          45
NodesListFmtToDataFmt ......completed !

ElemsListFmtToDataFmt ......
      100%    Total Elements:        100
ElemsListFmtToDataFmt ......completed !

NodesDataFmtToBinFmt ......
      100%       Total Nodes:          45
NodesDataFmtToBinFmt ......completed !

ElemsDataFmtToBinFmt ............
      100%    Total Elements:        100
ElemsDataFmtToBinFmt ...... Completed !

ReadElem ............
ReadElem ...... Completed !

ReadVert ............
ReadVert ...... Completed !

ElemIncMat......
Microsoft Pinyin 半 :
```

图 20-1. 程序运行过程

```
ReadRefElemInfo ........
ReadRefElemInfo Completed !



InitGlobPara ........
InitGlobPara Completed !



InitElem ........
InitElem Completed !


      0.036          0       -0.0253898417799042
      0.292         10         -8.01701791034494
      0.294         20         -10.4579565287161
      0.293         30         -11.6151998671721
      0.262         40         -9.68055765969053
      0.276         50         -11.2275480839813
      0.272         60         -14.3339145517565
      0.259         70         -12.8833587601213
Microsoft Pinyin 半 :
```

图 20-2.程序运行过程 续

程序的计算结果指定输出到 EvsT_E4H3PE.DAT 文件中。




```
5.833333333333e-011   39.2069274121217   45.8649323575863   43.612187327486   -1.09370471764853   -58.2279873570203
1.166666666667e-010   32.4206709326448   54.4309008688397   46.5714863109963   -1.0604140097179   45.9388209424605
1.75e-010   32.3761336713781   39.5905165781278   50.5495963165712   0.663454265685984   57.903173751171
2.333333333333e-010   40.3290166949974   17.4347800002087   57.3080747571154   0.893154801178472   16.1979738916179
2.916666666667e-010   51.3533065310073   8.38544785171579   61.2118995922169   0.236434126186572   10.4280158002295
3.5e-010   47.9127924813959   4.36170632828559   58.3083350919303   -0.426127711838448   -30.9717750859894
4.083333333333e-010   25.7302527090729   4.01054003258329   63.6699207718343   -0.249655118106793   3.30451862111612
4.666666666667e-010   8.30824232950546   17.7604194181142   73.0235086058764   0.484643353977852   30.7177413705244
5.25e-010   -9.29803544949786   26.400902279042   64.9500101662165   0.507594459456888   -18.6299391009414
5.833333333333e-010   -41.5306163206824   17.1068360343421   48.9934587161066   0.159224492439816   -25.1778994763998
6.416666666667e-010   -54.188694570913   10.48516997286   36.9399107387907   -0.180994665192943   5.03532375130672
7e-010   -28.04793141548690   12.5740298397369   15.0481187832256   -0.643235538106313   -5.64141881029713
7.583333333333e-010   -2.88359873656881   9.121527641786 56   -12.5438194104718   -0.610498528873534   -13.0681903683183
8.166666666667e-010   -2.72206471933149   0.486251427083097   -20.3681596651633   0.0710642782604732   15.4947148130748
8.75e-010   -10.4644545897566   -0.326695363611775   -2.869823356223641   0.48604887557786   32.7782180050509
9.333333333333e-010   -20.4254310229158   9.432509814453 52   21.7403532607829   0.329048004517792   31.7521875934 65
9.916666666667e-010   -35.4331367789988   14.7667089739883   23.2120573676227   0.0877554838449047   17.3078823043355
1.05e-009   -36.032565539237   1.49421029233265   3.58518323826718   -0.150450676584802   -27.3779157960275
1.108333333333e-009   -14.552330069929   -21.5086422484907   -28.4771042489576   -0.382726120790824   -57.7449680865921
1.166666666667e-009   -2.24448331081534   -33.4414290959605   -26.8150533139307   -0.332819989957413   -22.8620243449786
1.225e-009   -16.4621761401379   -28.8318407802336   9.02777452626593   -0.13947304208833   29.7149859542953
1.283333333333e-009   -27.1611367284275   -20.7261116767472   31.5001368764412   -0.0646789287973423   46.8832809482479
1.341666666667e-009   -10.8500563932319   -23.2883991566647   9.28215025612865   0.0609948913412845   38.2457703871879
1.4e-009   9.198984443865693   -36.2611809403367   -30.8850557003362   0.25900629469667   14.9407161227935
1.458333333333e-009   3.16261479525411   -46.2876271866155   -47.0924743148376   0.308509603029366   -17.8034545897398
1.516666666667e-009   -21.2751811633069   -44.6969881352553   -40.2742497950022   0.244688312914955   -32.7707614521348
1.575e-009   -34.4510185187321   -36.084466909682   -37.4170903270518   0.0842965918527633   -13.5226220707985
1.633333333333e-009   -24.6849351964781   -27.8000346055566   -40.7860492365378   -0.244801926220239   21.2297894779383
1.691666666667e-009   -8.1058613361954   -24.8031013093507   -34.1529651630919   -0.392150564398377   40.7391455908402
1.75e-009   -3.280318125374 26   -32.6268607237513   -21.4210303596368   -0.00656645679397524   27.7564015130347
1.808333333334e-009   -9.3980761834455 5   -49.1610855204976   -21.3298583318255   0.507835967520713   -9.13721983395525
1.866666666667e-009   -15.1053816251494   -59.9325388503229   -35.9668578517261   0.526233518826123   -40.4158712708158
1.925e-009   -16.4143243215515   -57.2578629595545   -50.4839695958814   0.0775027051427627   -38.689552413528
1.983333333334e-009   -16.0767932138414   -53.9277473084713   -53.6606578684745   -0.333024516830054   -0.717739733152428
2.041666666667e-009   -14.7233701925662   -61.79630734927   -44.2962398706603   -0.359534093969264   42.1280520213823
2.1e-009   -11.0755560848   -68.8288181740718   -27.8788587732254   -0.0932189776282629   42.8933473134284
2.158333333334e-009   -5.954693909949 26   -56.5476769934926   -15.3921682505521   0.0730793208924098   -3.89686868519093
2.216666666667e-009   -2.167922134348 04   -30.4407720214905   -17.1296388296136   -0.0690010244787195   -40.8526345795732
2.275e-009   -1.445302711283 21   -12.3109753771475   -32.5514481459865   -0.238776085445797   -21.0415432692533
2.333333333334e-009   -1.001972944949 53   -8.97945425693404   -49.7195184745609   -0.127052376793366   23.6987203317024
2.391666666667e-009   5.029359624638 08   -8.65474940572126   -55.1779385151554   0.070393156399695   32.4584648480746
2.45e-009   16.97480548010 07   -4.29460952727648   -43.3803541963272   0.00606413163410646   4.5981110165507
```

图 21. 部分计算结果

在 GPU 硬件为 GeForce GT 635M(notebook)的环境下，程序运行时间如图 22-1 所示。220

```
MatJacobi ........
MatJacobi Completed !



ReadRefElemInfo ........
ReadRefElemInfo Completed !



InitGlobPara ........
InitGlobPara Completed !



InitElem ........
InitElem Completed !

Time to generate:  29.4 min

Microsoft Pinyin 半 :
```

图 22-1 .在 GeForce GT 635M 环境下 GPU 版本程序的运行时间

在 CPU 硬件为 Intel® Core™ i5-3317U CPU@ 1.70GHz 的环境下，程序运行时间如图 22-2 所示。

```
     0.254     32550        -9.38108266661829
     0.254     32560         17.3188085545441
     0.249     32570         -10.671778877128
     0.267     32580        -9.73669619362234
      0.25     32590         23.6760387833922
     0.255     32600        -11.0379806004442
     0.251     32610        0.712635744625621
     0.251     32620         13.3878908042381
     0.249     32630        -7.29479942976498
     0.248     32640         -11.381243663793
     0.249     32650          14.878036151427
     0.256     32660        -17.6779460210238
     0.253     32670        -13.9730552299699
     0.255     32680         46.5346797784931
     0.252     32690        -30.6307459674109
     0.252     32700         3.02375532002969
     0.251     32710         10.4843925860135
      0.25     32720        -13.9472344476656
     0.254     32730         10.4263374180894
     0.252     32740         1.31089646100268
     0.249     32750        -5.00761846615144
     0.251     32760         12.6077442458788
Time to generate:  14 min

Microsoft Pinyin 半 :
```

图 22-2 .在 Intel® Core™ i5-3317U CPU@ 1.70GHz 环境下 CPU 版本程序的运行时间

从 2 张图的对比中可以看出，CPU 版本的运行速度更快。其原因主要有以下几种可能：

1. GPU 硬件性能有限。
2. 代码仍有优化的余地。比如 GPU 版本的程序中因为资源竞争的问题仍有使用原子写操作，而原子写操作开销是很大的。另外，分配给核函数的线程块数量和线程数量的不同都会极大地影响函数执行的速度。

3. 其他。

而在 GPU 硬件为 NVIDIA Tesla C2070 的环境下，程序运行时间如图 23-1 所示。

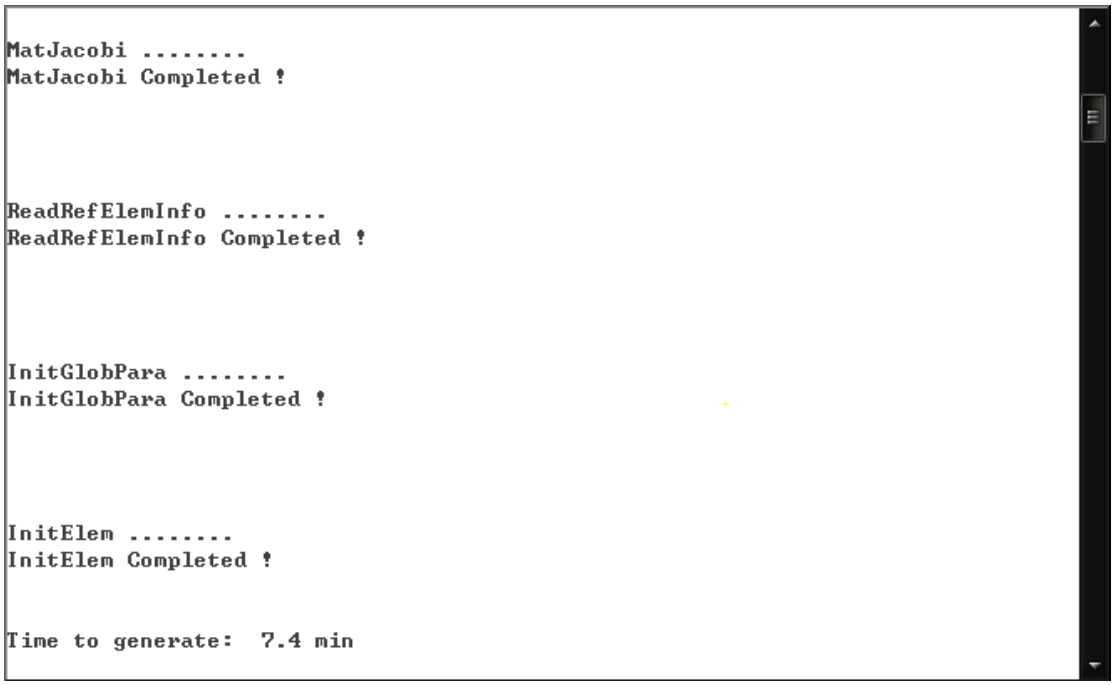


图 23-1 .在 NVIDIA Tesla C2070 环境下 GPU 版本程序的运行时间

在 CPU 硬件为 Intel® Xeon® CPU E5-2630 v2 @ 2.60GHz 2.60GHz 的环境下，程序运行时间如图 23-2 所示。

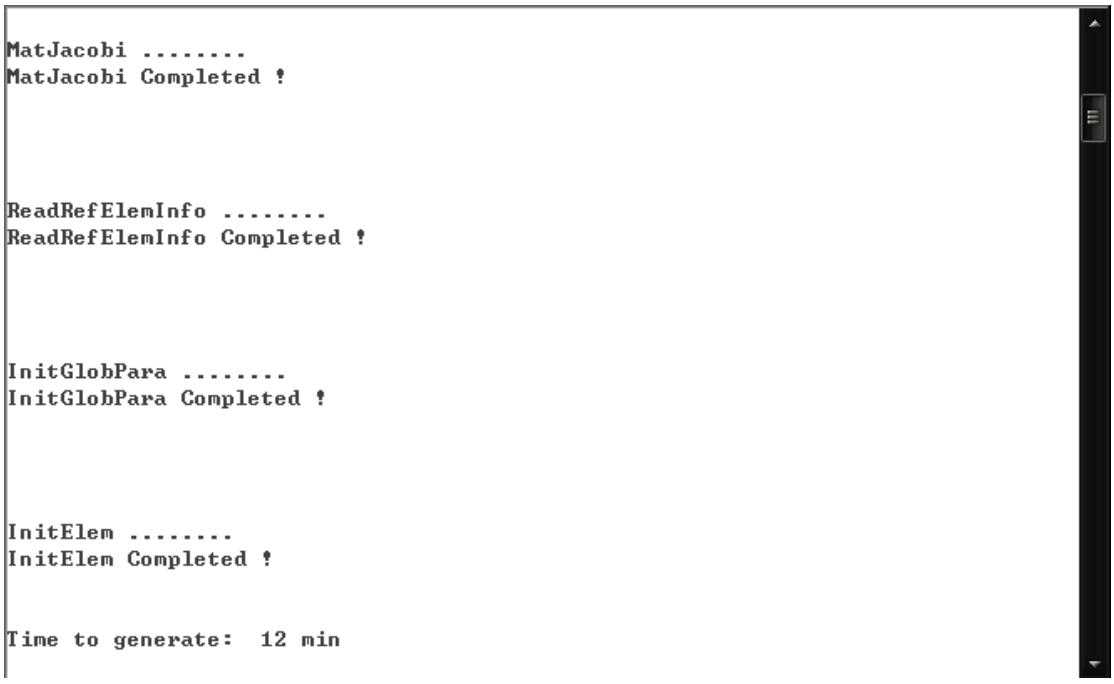


图 23-2 .在 Intel® Xeon® CPU E5-2630 v2 @ 2.60GHz 2.60GHz 环境下 CPU 版本程序的运行时间

从图 23-1 和图 23-2 这 2 张图的对比中可以看出，GPU 版本的运行速度更快。

一般来说，程序的运行速度受硬件环境的影响。GPU 硬件性能越高，计算速度越快。另外数据量的大小也会有影响。数据量越大，GPU 版本的程序越快，与 CPU 版本程序的差距会随着数据量的增大而越来越明显。另外，代码的优化也是非常关键的。

参考文献：

[1] 金建铭.电磁场有限元方法(美). 西安电子科技大学出版社.1998,1.

[2]齐红星，张杰.守恒无伪解麦克斯韦方程间断元研究（1）一维和两维情况.华东师范大学.2013,01,03.

[2] Jason Sanders. gpu 高性能编程 cuda 实战(美). 机械工业出版社.2011,1.

[3] NVIDIA. CUDA C Programming Guide - v7.5. 2015,9.