# GPU部分

## 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。

## 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1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。

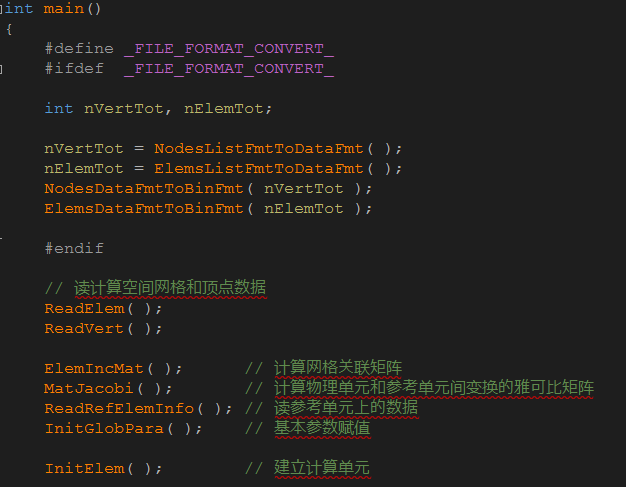


图1 计算开始前的准备工作

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。

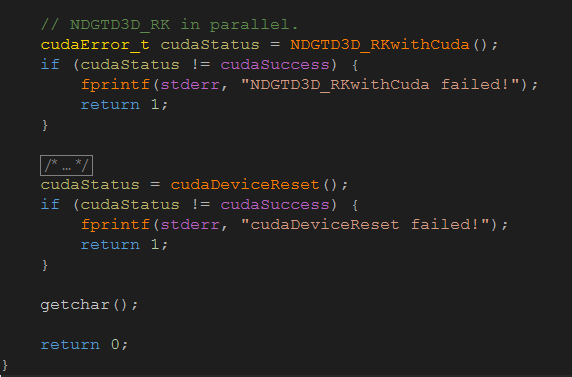


图2 函数NDGTD3D\_RKwithCuda()

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。

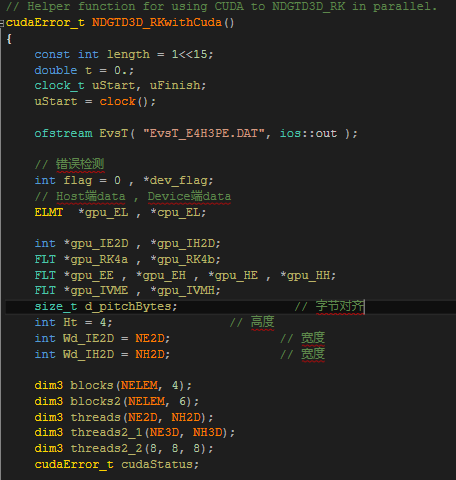


图3 变量声明以及初始化

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。

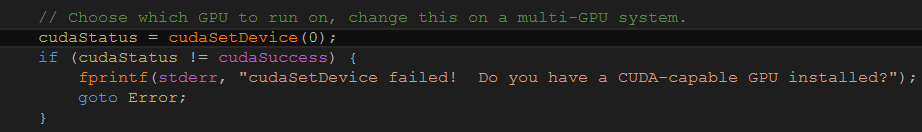


图4 选择GPU设备

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)

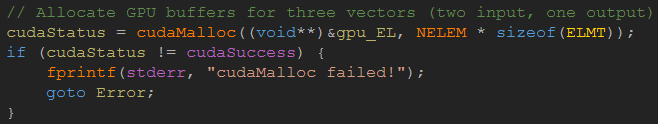


图5 分配一维GPU内存

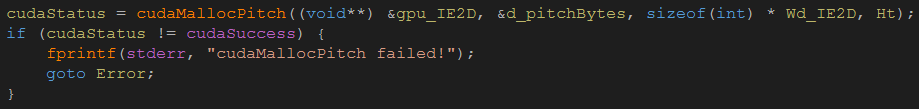


图6 分配二维GPU内存

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。

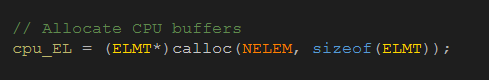


图7 分配CPU内存

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。

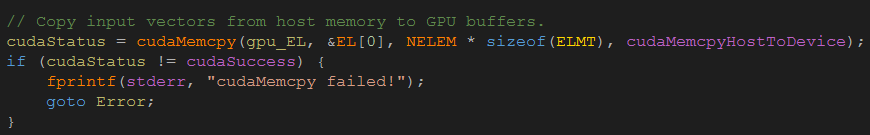


图8 将一维数据从CPU内存拷贝到GPU内存中

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。

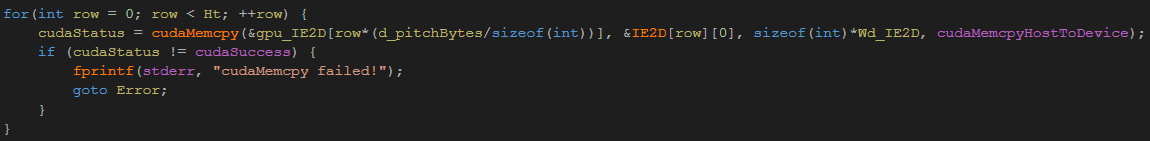


图9 将二维数据从CPU内存拷贝到GPU内存中

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);

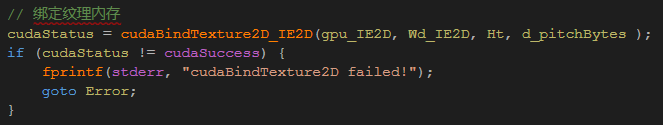


图10-1 绑定纹理内存

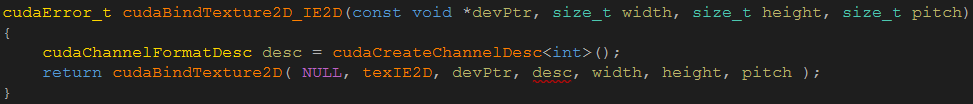


图10-2 绑定纹理内存 续

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))

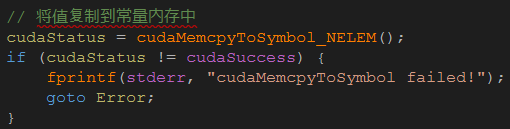


图11-1 将数据从CPU内存拷贝到GPU常量内存中

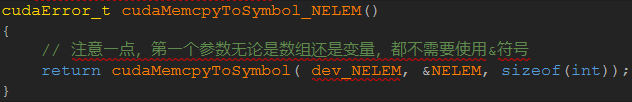


图11-2将数据从CPU内存拷贝到GPU常量内存中 续

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12的CPU版本下计算部分的简化代码。

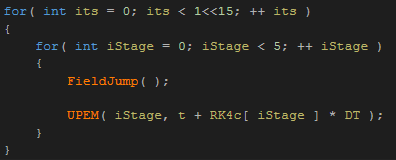


图12 CPU版本下计算部分的简化代码

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。

### 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 );

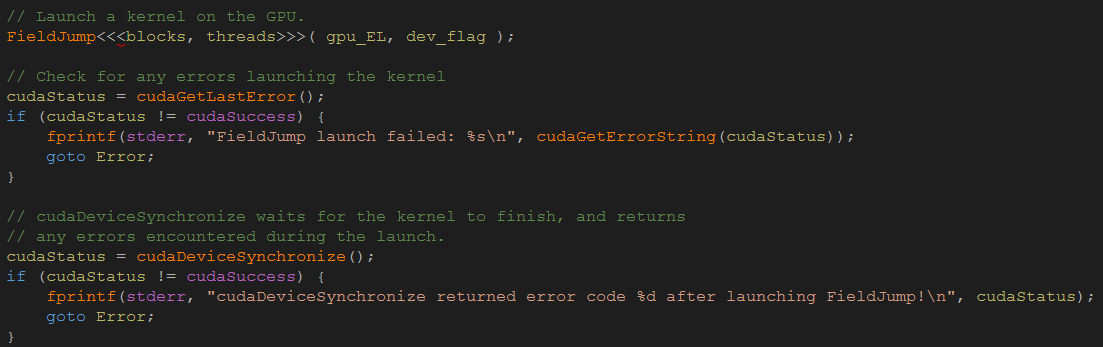


图13 GPU版本下FieldJump函数的调用

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。

|  |
| --- |
|  |
| 图(a) GPU版本的FieldJump函数部分代码 |
|  |
| 图(b) CPU版本的FieldJump函数部分代码 |

图14 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。

## 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14 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 );

## 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。

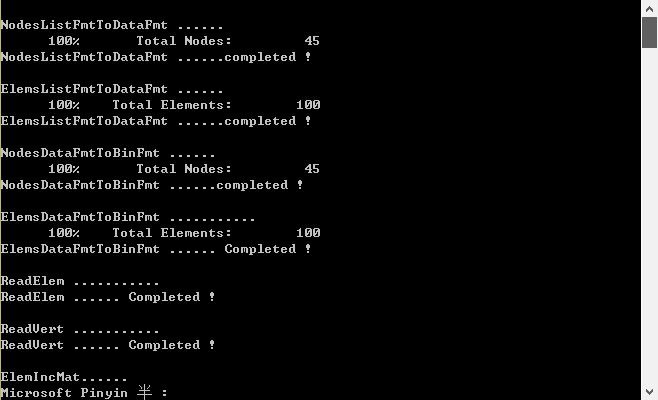


图15-1 程序运行过程

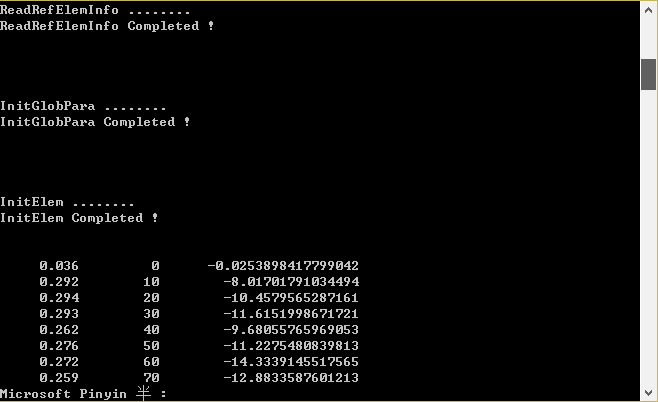


图15-2程序运行过程 续

程序的计算结果指定输出到EvsT\_E4H3PE.DAT文件中。

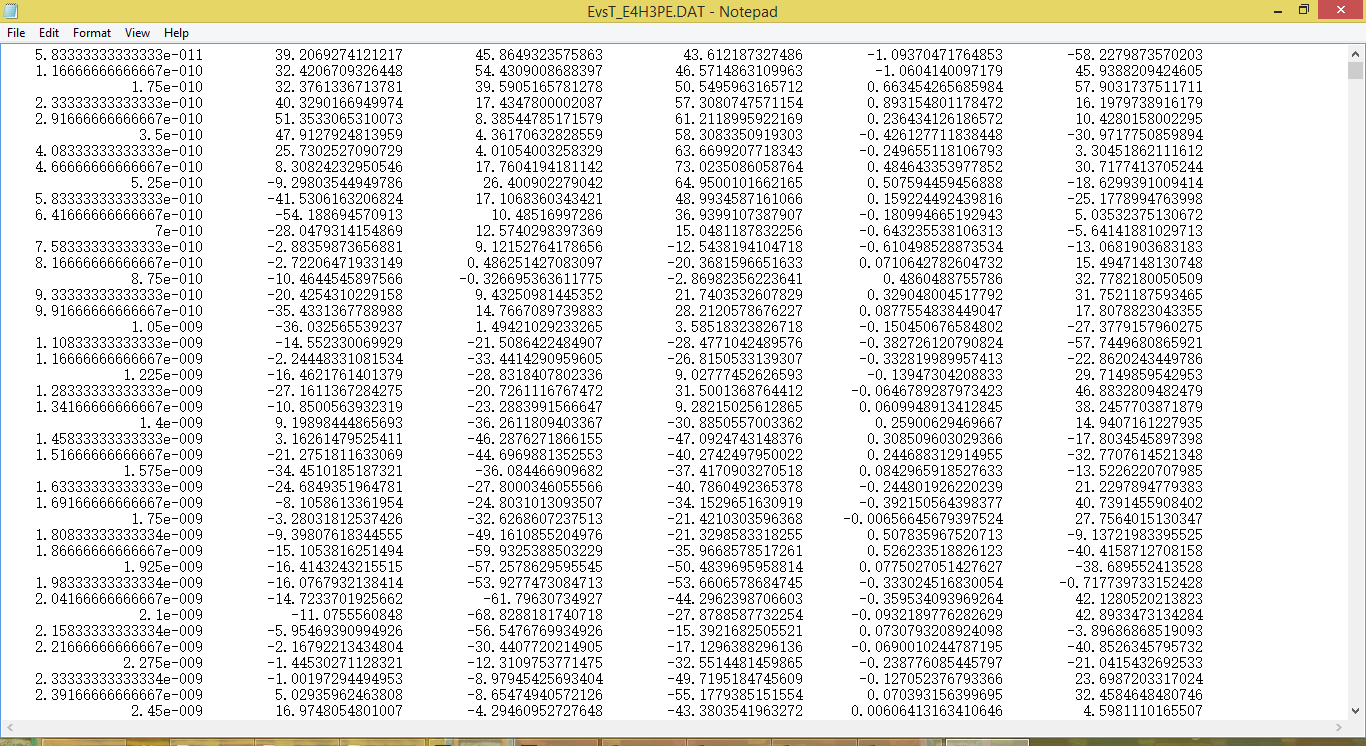


图16 部分计算结果

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。一般来说，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。

在GPU硬件为GeForce GT 635M(notebook)的环境下，程序运行时间如图17-1所示。

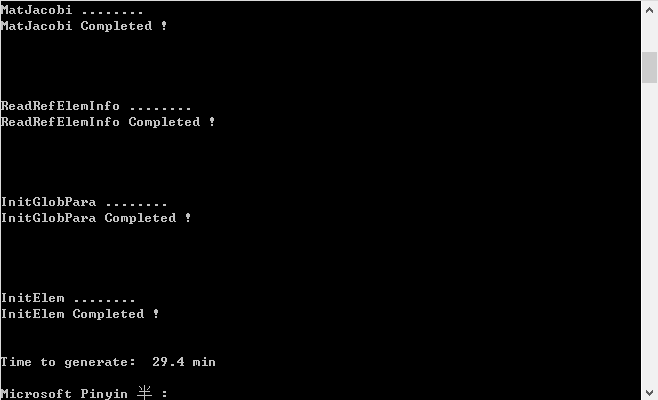


图17-1 在GeForce GT 635M环境下GPU版本程序的运行时间

在CPU硬件为Intel® Core™ i5-3317U CPU@ 1.70GHz的环境下，程序运行时间如图17-2所示。

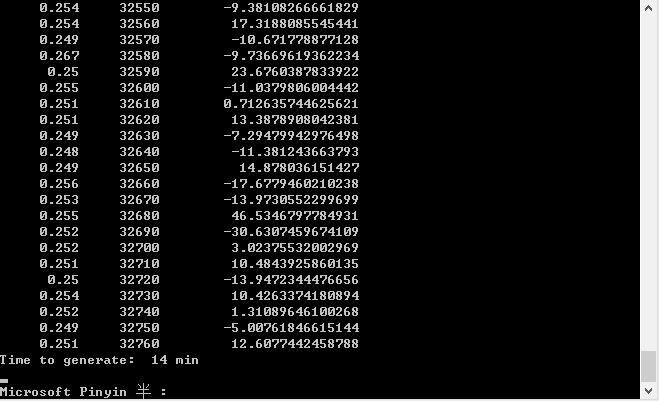


图17-2 在Intel® Core™ i5-3317U CPU@ 1.70GHz环境下CPU版本程序的运行时间

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。

## 6. 参考文献

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

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