郑海亮, 李兴东, 王哲, 等. 高分辨锥束 CT 并行重建算法在基于 NVDIA GPU 显卡计算平台上的实现[J]. CT 理论与应用研究, 2014, 23(5): 805-814.
Zheng HL, Li XD, Wang Z, et al. A practice on parallel reconstruction algorithm of high resolution cone beam micro-CT based on NVDIA GPU graphic card[J]. CT Theory and Applications, 2014, 23(5): 805-814.

高分辨锥束 CT 并行重建算法在基于 NVDIA GPU 显卡计算平台上的实现

郑海亮[1,3]，李兴东[2]✉，王哲[3]，魏存峰[3]，常彤[1]

1.首都医科大学附属北京世纪坛医院医学工程研究室，北京 100038
2.中国计量科学研究院电离辐射计量研究所，北京 100013
3.中国科学院高能物理研究所核辐射与核能技术重点实验室，北京 100049

摘要：目的：探讨高分辨率锥束显微 CT 断层重建中引入并行计算的必要性及其加速效果。方法：在具有并行计算功能的 GPU 显卡（NVIDIA QUADRO K5000 显卡，显存 4G）中为投影图像和重建体数据分配显存空间，每一个像素分配一个线程进行投影图像的各种校正和滤波，再给每个体素分配一线程进行反投影重建，在显存中实现全部断层重建。程序使用 C++面向对象方法实现，内核函数用 CUDA 实现。结果：重建体数据大小是 2 048×2 048×128，每个体素用 32 位浮点数记录，实验采集 1 800 张投影，每张投影图像大小为 2 048×1 536，重建时间小于 9 min，是图像采集时间的 1/3，是基于 CPU 重建耗时的 2%。将 GPU 并行重建得到的图像和 CPU 单线程重建图像结果进行对比，数据结果一致，满足实验设计的要求。结论：并行计算引入高分辨锥束 CT 重建可大大提高重建速度，并且能实现采集与重建同步进行。

关键词：锥束 CT；并行计算；高分辨率；显微 CT；重建速度；图像质量

文章编号：1004-4140（2014）05-0805-10 **中图分类号**：TN 919.8；TN 911.7 **文献标志码**：A

显微 CT 源于工业 CT，其特点是样品体积小、系统空间分辨率很高[1]。显微 CT 在揭示物质组成以及内部单元之间结构关系方面具有显著优势，在微电子工业、材料科学[2]、石油探勘[3]、环境科学[4]及生物医学和生命科学[5-6]等领域已经得到了广泛应用。近年来，微型计算机性能不断提升，探测器单元已小到数十微米，纳米级的 X 射线球管不断涌现，各种功能强大的显微 CT 设备在实验室和企业陆续问世。

虽如此，目前显微 CT 扫描和重建速度要远慢于医学 CT，制约了其在医学上的应用。显微 CT 成像速度慢，所需处理数据量太大是原因之一。医学 CT 上应用最多的是 512×512 重建矩阵，而显微 CT 多用 1 024×1 024，甚至是 2 048×2 048 重建矩阵。以我们的经验为例，探测器 1 024×1 024 矩阵，扫描 720 张投影图并重建耗时可达到 20 多分钟，相比之下医学 CT 完成一次头部扫描仅需数秒，可见两种 CT 在成像速度的差异显著。采用响应时间更短读出更快的探测器可显著提高采集时间；但耗时与重建矩阵大小呈指数关系，而基于 CPU 重

收稿日期：2014-07-24。
基金项目：国家自然科学基金（11005119）；北京世纪坛医院院级课题（2013-C11）；中国计量科学研究院院级课题（25-JB1336-13）。

建的处理模式是串行方式，因此仅仅提高 CPU 主频对重建速度提升效果甚微，相比之下多路并行计算是应对上述问题一个很好的解决办法[7−8]。有研究表明，采用平行计算的重建方式可提速 24 倍[9]。

重建构架是 CT 系统的灵魂，采用多路并行计算，其重建架构也有别于串行计算特征[8−11]。从理论上说，同步扫描重建模式可以在多路并行架构实现。在该架构下，探测器的效能可得到最大程度的发挥，对于推广先进探测器技术应用也有重要意义。

多孔介质流体动力学是水文地质和环境等学科所关注的重要内容，在这些领域的研究中石英砂常作为多孔介质的实验模拟材料，用于模拟多孔介质的孔隙结构，流体的赋存状态[4]和流动现象。本研究用显微 CT 对石英砂样品进行扫描，分别用基于 CPU 和 GPU 的重建算法进行了重建，并比较重建时间和图像质量，一方面可以为多孔介质流体动力学研究提供一种新思路，同时也旨在探讨 GPU 并行计算技术对显微 CT 重建提速的实践意义

1 原理与方法

CT 图像重建原理源于 X 射线通过介质时衰减的物理规律，根据扫描所获得的投影值来求解断层上衰减系数的分布。理想单能窄束 X 射线投射各向同性均匀连续介质时，强度衰减的物理规律符合朗伯（Lambert）定律：

$$I(x,z)=I_0\exp\left(-\int_L \mu(x,y,z)\,\mathrm{d}s\right)$$

式中，I_0 是入射 X 射线的强度，I 是通过厚度为 x 的均匀介质后 X 射线的出射强度，μ 是均匀介质的线性衰减系数。

Feldkamp 锥形束图像重建算法是专门为锥形束的圆形焦点轨道而设计的[12]，其实质上是一种个先滤波后反投影的 FBP 算法，当锥形束张角比较小时，重建图像与平行束重建结果接近[13]。

1.1 FDK 算法原理

FDK 算法依次包含预处理、几何伪影校正、衰减系数线性积分[13]。

预处理：将内存中的投影数据拷贝到显存中，如果投影图像的 Y 轴方向不与旋转轴平行，那么将投影图像旋转 90°，然后将投影数据转换成 0 到 1 之间的浮点数。

几何伪影校正：将投影图像数据沿水平方向移位，再沿竖直方向移位，让 X 射线焦点对准投影图像的中心，然后投影数据绕投影中心进行旋转校正，让投影图像的竖直中间线和样品旋转轴对齐。

衰减系数的线积分：$R_\beta(p,q)$ 衰减系数在投影路径 L 上的线积分，

$$R_\beta(p,q)=\ln\left(\frac{I_0}{I(p,q)}\right)=\int_L \mu(x,y,z)\,\mathrm{d}s$$

其中，I_0 是入射 X 射线的强度，$I(p,q)$是通过一定厚度为 L 的均匀介质后 X 射线的出射强度，(p,q)是投影到探测器的位置，$\mu(x,y,z)$是均匀介质在空间位置为 (x,y,z) 的线性衰减系数。

折算系数：投影数据乘上一个系数 $\dfrac{D}{\left(D^2+p^2+q^2\right)^{\frac{1}{2}}}$

$$R_\beta'(p,q)=\frac{D}{\left(D^2+p^2+q^2\right)^{\frac{1}{2}}}R_\beta(p,q)$$

$R_\beta(p,q)$ 是采集的投影图像衰减系数的线积分，p 和 q 是投影像素的位置，q 平行旋转轴，p 垂直 q，D 是射线源到旋转中心的距离。

图像滤波：沿着图像水平方向进行傅立叶变换，

$$R_\beta'(u,q)=\mathrm{FFT}\left\{R_\beta'(p,q)\right\}$$

用 RamLak 滤波函数 $\varpi(u)$ (见图 1)，进行滤波，

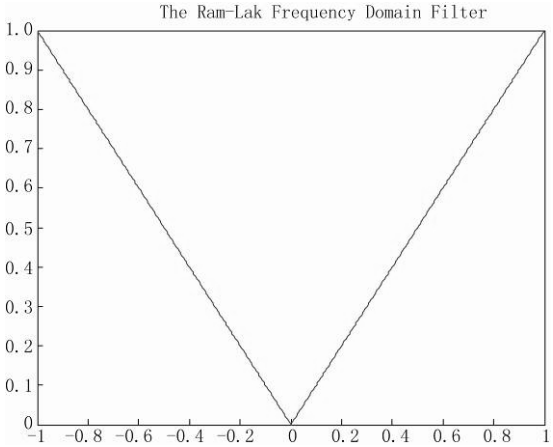


图 1　RamLak 滤波器

Fig.1　Ramlak filer

然后进行傅立叶反变换。

$$Q_\beta(p,q)=\mathrm{IFFT}\left\{R_\beta'(u,q)\left|\varpi(u)\right|\right\}$$

反投影：将特定旋转角度对应的投影像素值累加回体数据。

投影图像的像素坐标(p,q)和对应射线路径上的体素坐标 (t,z)，它们之间必须通过系数 $\dfrac{D}{D-s}$ 统一起来，因此 $p=\dfrac{D}{D-s}t$，　$q=\dfrac{D}{D-s}z$

$$f_b(t,s,z)=\int_0^{2\pi}\frac{D^2}{(D-s)^2}Q_\beta\left(\frac{D}{D-s}t,\frac{D}{D-s}z\right)\mathrm{d}\beta$$

其中 $f_b(t,s,z)$ 是反投影的体素值，(t,s,z) 是体素的索引值。

体数据灰度调整：将体数据映射到 0 到 1 之间的浮点。

$$f(t,s,z)=\begin{cases} f_{\min}, & f_b(t,s,z)<f_{\min} \\ \dfrac{f_b(t,s,z)-f_{\min}}{f_{\max}-f_{\min}}, & f_{\min}<f_b(t,s,z)<f_{\max} \\ f_{\max}, & f_b(t,s,z)>f_{\max} \end{cases}$$

1.2 并行算法的实现

采用传统的基于 CPU 的重建（串行算法）实现 FDK 算法计算耗时超过 24 小时。我们通过算法分析后发现，在对投影图像像素和重建体数据体素进行处理时，各个像素和体素的计算相对独立，非常适合并行计算。并行计算的硬件支持为具有并行计算功能显卡的 GPU，它采用单指令流多数据流（Single Instruction Multiple Data，SIMD）的并行执行模式，图形处理器（GPU）拥有强大的并发处理能力和可编程流水线，面对单指令流多数据流时，运算能力远超过传统 CPU。OpenCL 和 CUDA 是目前最广泛使用的两种图像并行处理的函数库[9−10]，这里我们选用 CUDA 并行开发工具。

投影数据的每一个像素作为一个独立的运算单元，分配一个线程，通过 CPU 发出一个指令，让 GPU 对所有像素同时进行处理。对体数据的每一个体素作为一个独立的运算单元，分配一个线程，通过 CPU 发出一个指令，让 GPU 对所有体素同时进行处理。这样实现投影图像和重建体数据的并行处理。

```
定义每个线程块里有 32×32 个线程
#define BLOCK_DIM_Y  32
#define BLOCK_DIM_X  32
threads = dim3( BLOCK_DIM_X, BLOCK_DIM_Y, 1 );

投影数据的线程块数
grid   = dim3( width/BLOCK_DIM_X,  height/BLOCK_DIM_Y,  1 );
体数据的线程块数
grid   = dim3( width/BLOCK_DIM_X,  height/BLOCK_DIM_Y,  frame );
```

CUDA 多线程优化和 CPU 单线程编程方式的不同，如表 1。

表 1 并行和串行编程模式的比较
Table 1 A comparison of parallel and serial programming mode

	GPU	CPU

2D	i = blockIdx.y * blockDim.y + threadIdx.y;
	j = blockIdx.x * blockDim.x + threadIdx.x;
	if(i<height && j<width) {
	}
		for(i=0; i<height; i++) {
		for(j=0; j<width; j++) {
		}
		}

3D	i = blockIdx.z;
	j = blockIdx.y * blockDim.y + threadIdx.y;
	k = blockIdx.x * blockDim.x + threadIdx.x;
	if(i<frame && j<height && k<width) {
	}
		for(i=0; i<frame; i++) {
		for(j=0; j<height; j++) {
		for(k=0; k<width; j++) {
		}
		}
		}

注：这里 i，j，k 表示索引值。

1.2.1　分配存储空间

在内存中分配整个投影序列的存储空间，用于将所有的投影数据导入到内存中，分配与显存大小相等的内存空间，用于存放由显存中拷贝到内存的体数据。在显存中分配一帧投影数据的存储空间，用于投影数据的各种校正，分配一帧投影数据的滤波存储空间，用于投影数据的频域滤波，然后分配体数据的存储空间。

1.2.2　数据流实现

首先将采集到的投影序列拷贝到内存中，将每张投影图依次送入显存，完成投影数据的校正、滤波和反投影，等到全部的投影图像使用完成后，对反投影的体数据进行灰度调整后，将显存中的体数据拷贝回内存中，最后存盘。

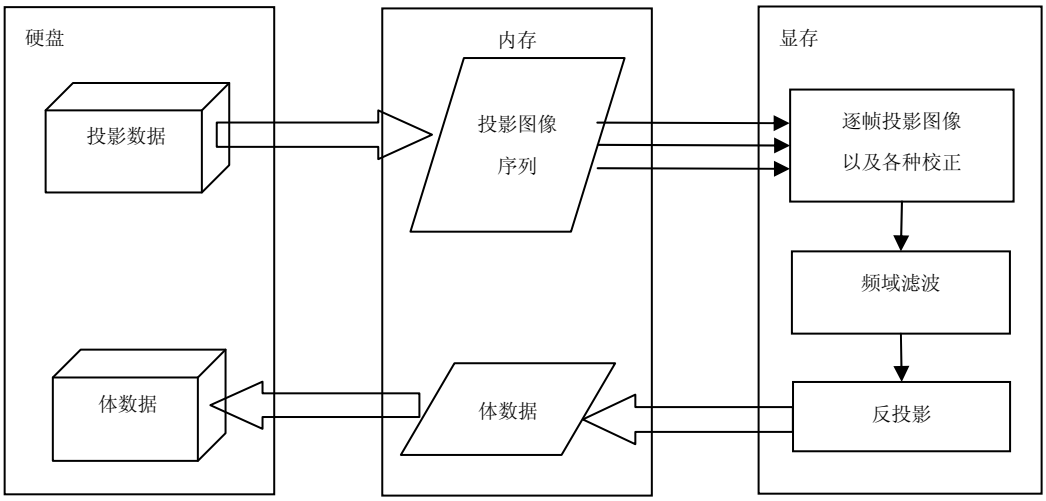


图 2　图像重建的数据流程简图

Fig.2　Schematic diagram of data flow

1.2.3　程序结构

程序的架构使用 C++面向对象的方法，内核函数用 CUDA 实现。将投影数据的处理，频域滤波和反投影体数据分别封装到投影图像类，滤波类和体数据类中。对于在 CPU 中执行的代码，函数名后面添加“_CPU”后缀；对于在 GPU 中执行的代码，函数名后面添加“_GPU”后缀。通过一个没有后缀的函数，函数里面的 if.....else..... 的判断，如果 m_bGPU=true，将执行 GPU 的并行运算，否则执行 CPU 的串行运算，这样将 CPU 和 GPU 的执行过程分开。需要在 GPU 中执行的内核函数的代码，在.cu 文件中实现，内核函数前面要加__global__关键字，说明该函数是在 GPU 的内核函数中执行。

（1）投影图像类

在 ProjectImage.h 头文件中

```
class CProjectImage
{
protected:
    bool m_bGPU;          // 是否使用 GPU
```

```
    int m_Height;           // 图像的高度
    int m_Width;            // 图像的宽度
    float* m_pData;          // 图像数据的存储地址

    void Process_CPU();     // 在内存中实现的代码
    void Process_GPU();     // 在显存中实现的代码
public:
    void Process();
};
```

在 ProjectImage.cpp 文件中实现的代码

```
void CProjectImage::Process()
{
    if (m_bGPU) {
        Process_GPU();
    }else {
        Process_CPU();
    }
}
```

在 ProjectImage_kernel.cu 文件中实现 GPU 并行代码

```
void CProjectImage::Process_GPU()
{
    Process_Kernel<<< grid, threads >>>( m_pData, m_Height, m_Width );
}
```

其中内核函数

```
__global__ void Process_Kernel( float* d_Data, int height, int width )
{
    // 内核函数实现的代码
}
```

（2）体数据类

在 Volume.h 头文件中

```
class CVolume
{
protected:
    bool m_bGPU;          // 是否使用 GPU
    int m_Frame;           // 体数据帧数
    int m_Height;           // 体数据高度
    int m_Width;           // 体数据宽度
```

```
    float* m_pData;          // 体数据存储地址

    void BackProject_CPU( CProjectImage* pProjectImage );   // 在内存中实现的代
码
    void BackProject_GPU( CProjectImage* pProjectImage );   // 在显存中实现的代
码
public:
    void BackProject( CProjectImage* pProjectImage );
};
```

在 Volume.cpp 文件中实现的代码

```
void CVolume::BackProject( CProjectImage* pProjectImage )
{
    if (m_bGPU) {
        BackProject_GPU( CProjectImage* pProjectImage );
    }else {
        BackProject_CPU( CProjectImage* pProjectImage );
    }
}
```

在 Volume_kernel.cu 文件中实现 GPU 并行代码

```
void CVolume::BackProject_GPU( CProjectImage* pProjectImage )
{
    int projH = pProjectImage->m_Height;
    int projW = pProjectImage->m_Width;
    float* pProjData = pProjectImage->m_pData;
    BackProject_Kernel<<< grid, threads >>>( m_pData, m_Height, m_Width,
                                        pProjData, projH, projW );
}
```

其中内核函数

```
__global__ void BackProject_Kernel( float* d_volData, int volF, int volH, int volW,
                                    float* d_projData, int projH, int projW )
{
    //内核函数实现的代码
}
```

1.3　实验软硬件环境

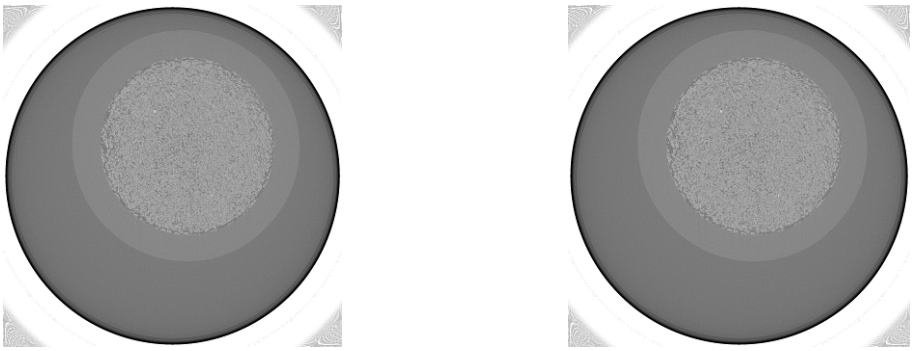
实验样品为一根外径为 5 mm，内径为 4 mm 的聚碳酸酯（PC）管，里面装满 200～230 目的石英砂。

射线源焦点到旋转中心的距离是 29 mm，射线源焦点到平板探测器的距离是 866 mm，探

测器大小是 40 cm×30 cm，探测器的像素数是 2 048×1 536。载物台旋转 360°，采集 1 800 张投影图像，平均 0.2° 采集一张投影图。实验采用 NVIDIA QUADRO K5000 显卡作显存 4G。实验机的 CPU 为 Intel Core i7 4770 3.4 G 硬盘为 Seagate 1T，7200 转，缓存 64 MB，SATA3 接口，传输速度为 6 Gb/s；内存为 Kingston DDR3 1600 8 G。Win7 64 位操作系统，Microsoft Visual studio 2012 作为开发环境，调用 NVDIA 公司提供的图像处理函数库 CUDA 5.5。

2 结果与讨论

基于 CPU 和 GPU 重建断层图如图 3 所示。



（a）基于 CPU 重建断层图　　（b）基于 GPU 重建断层图

图 3 重建的中间断层切片，图中切片大小 2 048×2 048，其中的内切圆是重建区域，像素大小 6.5 μm×6.5 μm，层间距 4.9 μm

Fig.3 A section picture reconstructed, picture size 2 048×2 048, and pixel size 6.5 μm×6.5 μm, layer distance 4.9 μm

2.1 重建时间的比较

重建体数据大小是 2 048×2 048×128，每个体素是 32 bit 的浮点数，投影图像大小 2 048×1 536，共 1 800 张，重建时间在 9 min 内，是图像采集时间的 1/3，是基于 CPU 重建耗时的 2%。

2.2 图像质量比较

分别从 GPU 并行重建图像和 CPU 单线程重建图像的中间取 1 024×1 024 的像素数据进行对比，先求它们的像素平均值，

$$E(I_{\mathrm{GPU}})=\frac{1}{MN}\sum_{i=0}^{i=M-1}\sum_{j=0}^{j=N-1}I_{\mathrm{GPU}}[i,j]$$

$$E(I_{\mathrm{CPU}})=\frac{1}{MN}\sum_{i=0}^{i=M-1}\sum_{j=0}^{j=N-1}I_{\mathrm{CPU}}[i,j]$$

其方差为：

$$\sigma^2\left(I_{\mathrm{GPU}}\right)=\frac{1}{MN}\sum_{i=0}^{i=M-1}\sum_{j=0}^{j=N-1}\left(I_{\mathrm{GPU}}[i,j]-E\left(I_{\mathrm{GPU}}\right)\right)^2$$

$$\sigma^2\left(I_{\mathrm{CPU}}\right)=\frac{1}{MN}\sum_{i=0}^{i=M-1}\sum_{j=0}^{j=N-1}\left(I_{\mathrm{CPU}}[i,j]-E\left(I_{\mathrm{CPU}}\right)\right)^2$$

分析的结果是$\sigma^2\left(I_{\mathrm{GPU}}\right)=\sigma^2\left(I_{\mathrm{CPU}}\right)$，说明数据结果一致，两种重建方法的结果相同。将两幅图像做差得到灰度值全部为零的图像，也说明两种重建方式得到的图像结果是一致的，限于篇幅文中做差的图不再文中给出。

3　结论

本实验平台上已经实现了 6.5 μm×6.5 μm，层间距是 4.9 μm 的空间分辨能力，接近于纳米级水平，重建速度在 0.2 frame/s，相信这样的显微 CT 平台将成为研究多孔介质孔隙特征和多孔介质流体动力学的有力工具。

在本实验条件下图像重建时间在 9 min 内，是图像采集时间的 1/3，是基于 CPU 重建耗时的 2%。

将并行计算引入锥束 CT 重建大大提高重建速度，并且能实现采集与重建同步进行。采用面向对象的编程模式，将投影图像的大小，体数据的大小，以及存放数据的地址封装在类中，减少了全局变量的使用和参数的传递，便于维护。

参考文献

[1] Landis EN, Keane DT. X-ray microtomography[J]. Materials Characterization, 2010, 61(12): 1305-1316.
[2] 傅健, 魏东波, 李斌, 等. 显微 CT 技术研究进展及在材料科学中的应用[C]//第二届全国背散射电子衍射（EBSD）技术及其应用学术会议暨第六届全国材料科学与图像科技学术会议, 包头: 2007.
[3] Arns CH. A comparison of pore size distributions derived by NMR and X-ray-CT techniques[J]. Physica A: Statistical Mechanics and its Applications. 2004, 339(1/2): 159-165.
[4] Wu D, Peng XF. Saturation evolution induced by inner pore structural effects in a porous material during wetting[J]. International Journal of Heat and Mass Transfer, 2009, 52(19/20): 4664-4668.
[5] Mizutani R, Suzuki Y. X-ray microtomography in biology[J]. Micron, 2012, 43(2/3): 104-115.
[6] Clark DP, Badea CT. Micro-CT of rodents: State-of-the-art and future perspectives[J]. Physica Medica, 2014, 30(6): 619-634.
[7] Scherl H, Kowarschik M, Hofmann HG, et al. Evaluation of state-of-the-art hardware architectures for fast cone-beam CT reconstruction[J]. Parallel Computing, 2012, 38(3): 111-124.
[8] Noël PB, Walczak AM, Xu JH, et al. GPU-based cone beam computed tomography[J]. Computer Methods and Programs in Biomedicine, 2009, 98(3): 271-277.
[9] Okitsu Y, Ino F, Hagihara K. High-performance cone beam reconstruction using CUDA compatible GPUs[J]. Parallel Computing, 2010, 36(2/3): 129-141.
[10] Flores LA, Vidal V, Mayo P, et al. CT Image Reconstruction Based on GPUs[J]. Procedia Computer Science, 2013, 18: 1412-1420.
[11] Flores LA, Vidal V, Mayo P, et al. Parallel CT image reconstruction based on GPUs[J]. Radiation Physics and Chemistry, 2014, 95: 247-250.

[12] Feldkamp L, Davis L, Kress J. Practical cone-beam algorithm[J]. Journal of the Optical Society of America A-Optics Image Science and Vision, 1984, 1(6): 612-619.
[13] 曾更生. 医学图像重建[M]. 北京: 高等教育出版社, 2009.
[14] NVIDIA. CUDA C Programming Guide[EB/OL]. [2013-05]. http://www.nvidia.com/cuda.

A Practice on Parallel Reconstruction Algorithm of High Resolution Cone Beam Micro-CT Based on NVDIA GPU Graphic Card

ZHENG Hai-liang[1,3], LI Xing-dong[2✉], WANG Zhe[3], WEI Cun-feng[3], CHANG Tong[1]

1.Shijitian Hospital of Capital University of Medical Sciences, Beijing 100038, China
2.National Institute of Metrology, Beijing 100013, China
3.Key Laboratory of Nuclear Radiation and Nuclear Energy Technology, Institute of High Energy Physics, Chinese Academy of Sciences, Beijing 100049, China

Abstract: Objective: To explore the feasibility of parallel computing applying in high-resolution cone beam micro-CT reconstruction and its impact on reconstruction speed. Method: Allocating video memory to projection pictures and reconstruction voxels in GPU graphic card (NVIDIA QUADRO K5 000, Video Memory 4G) with parallel computing, and allocating thread to each pixel for adjusting and filtering projection picture, and then allocating thread to each voxel for Back Projection, thus, all section reconstruction is implemented in graphic card. Result: Less than 9 minutes spent for 2 048 × 2 048 × 128 pixel matrix reconstruction, which is equal to 1/3 of data gathering time and 2% of CPU based reconstruction, under the condition that one voxel is recorded by 32 float, and each projection picture size is 2 048 × 1 536, and 1 800 projections are gained in one scanning. Conclusion: Parallel computing applied in cone beam CT reconstruction can greatly increase reconstruction speed and data gathering can simultaneously operate with reconstruction.

Key words: cone beam CT; parallel computing; high resolution; micro-CT; reconstruction speed; image quality

作者简介：郑海亮（1979－），男，2011 年中国科学院高能物理研究所博士后出站，2011 年至今在北京世纪坛医院医学工程研究室工作，从事医学影像理论与应用研究工作，E-mail：doczheng@126.com；李兴东✉（1957－），男，副研究员，东北师范大学物理系毕业，1991 年 3 月获日本国丰桥技术科学大学放射线计测工学硕士学位，1997 年至今在中国计量科学研究院电离辐射所工作，从事电离辐射剂量学以及电离辐射影像学的研究，现任辐射剂量学实验室主任。