大规模稀疏线性方程组的 GMRES-GPU 快速求解算法

柳有权[1,2)]，尹康学[1)]，吴恩华[2,3)]

[1)](长安大学信息工程学院 西安 710064)

[2)](中国科学院软件研究所计算机科学国家重点实验室 北京 100190)

[3)](澳门大学科技学院 澳门)

(youquan@chd.edu.cn)

摘 要：重开始广义极小残量法(GMRES)是求解大规模线性方程组的常用算法之一，具有收敛速度快、稳定性好等优点.文中基于 CUDA 将 GMRES 算法在 GPU 上进行并行算法实现，尤其针对稀疏矩阵矢量乘法运算，通过合并访问和共享内存策略相结合的手段使得算法效率大幅度提升.对于大规模数据集，在 GeForce GTX 260 上的运行结果相对于 Intel Core 2 Quad CPU Q9400@2.66 GHz 得到了平均 40 余倍的加速效果，相对于 Intel Core i7 CPU 920@2.67 GHz 也可得到平均 20 余倍的加速效果.

关键词：CUDA；GPGPU；重开始广义极小残量法；稀疏矩阵矢量乘法

中图法分类号：TP391

Fast GMRES-GPU Solver for Large Scale Sparse Linear Systems

Liu Youquan[1,2)], Yin Kangxue[1)], and Wu Enhua[2,3)]

[1)](*School of Information Engineering, Chang'an University, Xi'an* 710064)

[2)](*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190)

[3)](*Faculty of Science and Technology, University of Macau, Macao*)

Abstract: As a popular iterative method to solve linear equations, restarted generalized minimal residual method (GMRES) has the advantages of fast convergence and good stability. This paper implements a parallel GMRES in GPU based on CUDA. Particularly, the sparse matrix vector multiplication is optimized with coherence visiting and shared memory, which significantly improves the performance. We tested the paralleled GMRES on a GPU of GeForce GTX260, and compared its performance with those of the traditional GMRES on Intel Core 2 Quad CPU Q9400@2.66GHz and Intel Core i7 CPU 920@2.67GHz, which showed 40 times of speed-up and 20 times of speed-up on average respectively.

Key words: CUDA; GPGPU; generalized minimal residual method; sparse matrix vector multiplication

计算机图形学领域和一些实际工程应用中存在很多偏微分方程的求解，如软体变形、流体仿真、几何处理，这些方程在经过离散化后都转化成线性方程组，从而将复杂问题求解变成一个可计算机求解的问题.这类求解通常采用迭代法进行数值计算，因此此类方法的高效求解对这种复杂问题有着非常重要的意义.该线性方程组可统一表示为 $\boldsymbol{Ax}=\boldsymbol{b}$；其中 $\boldsymbol{A}$ 为 $n\times n$ 大小的系数矩阵，$\boldsymbol{x}$ 为 n 元变量，$\boldsymbol{b}$ 为已知

收稿日期：2010-09-25；修回日期：2010-11-30. **基金项目**：国家自然科学基金(60973066，60833007)；中国科学院软件研究所计算机科学国家重点实验室开放基金(SYSKF1004). **柳有权**(1976—)，男，博士，副教授，CCF 会员，主要研究方向为计算机图形学、虚拟现实. **尹康学**(1990—)，男，在校学生. **吴恩华**(1947—)，男，博士，教授，博士生导师，CCF 高级会员，主要研究方向为计算机图形学、虚拟现实.

量.在目前诸多求解大规模线性方程组问题的迭代法中,重开始广义极小残量法(generalized minimal residual method, GMRES)①[1-2]是很受欢迎的算法之一,它通过 Krylov 子空间矢量的最小残量来迭代求解,具有收敛速度快、稳定性好等优点.

目前有很多研究人员侧重于改进 GMRES 算法,以进一步提高该方法迭代的效率.如全忠等[3]通过构造多项式预处理因子来克服 GMRES 算法有时收敛很慢或停滞的缺陷,Habu 等[4]通过调整重新开始来加快 GMRES 的收敛速度.

但除了从算法层面来改进整体收敛速度,单步计算的效率提升同样重要,这一方面依靠硬件的计算能力的提升,另外新的计算架构通过对算法进行并行化处理也可提升算法计算效率.传统的高性能计算依赖大型机和计算集群,然而这样的计算系统都很昂贵.GPU 的发展为高性能计算提供了另外一种思路[5-6],它采用众核架构,即芯片上集成了多个并行处理单元.随着可编程性的出现,GPU 从单纯的图形流水线渲染转向通用计算上的应用(GPGPU),现在在高性能计算机领域也开始崭露头角.GPU 单位计算成本的下降引起了很多研究机构和企业的广泛重视,如国产天河系统由于采用 CPU+GPU 以较低的代价获得非常高的性能,在最近的全球高性能计算机排行榜上名列第一.NVIDIA[7]推出的 CUDA(compute unified device architecture)架构由于编程方式的革新更是推动了 GPU 芯片在高性能计算上的应用.

目前已有一些 GMRES 利用 CUDA 进行加速的工作,如 Wang 等在 GeFore GTX280 图形卡上获得 20 倍加速[8],Velamparambil 等类似的工作在 GeForce 8800 上获得 13 倍的加速[9],Ghaemian 等基于 NVIDIA Tesla C870 GPU 只获得 60%的加速[10].文献[11]对作为各种迭代求解方法都要使用的核心算法——稀疏矩阵与矢量乘法运算做了详细分析.虽然上述工作各自的硬件平台略有不同,但整体加速效率仍有提升的空间.

本文基于 CUDA 将 GMRES 算法在 GPU 上进行重新设计,尤其是对稀疏矩阵与矢量乘法部分,通过合并访问和共享内存的分配来优化负载;并充分利用 GPU 的众核处理能力,使得算法效率相对于 CPU 算法有大幅度提升.另外,通过跟 NVIDIA 提供的稀疏矩阵与矢量相乘的 GPU 算法[11]比较,本文算法实现相对于 NVIDIA 自身的代码也有好几倍的效率提升,同时代码在 http://imlab.chd.edu.cn/cuda/上公开,供研究人员免费使用.

1 GMRES 算法分析

关于 GMRES 算法的详细描述请参考文献[2],本文为了完整起见,对其稍作介绍.对于线性方程组 $\boldsymbol{Ax}=\boldsymbol{b}$,GMRES 算法的 m 阶 Krylov 子空间为 $\boldsymbol{K}_m=\mathrm{span}(\boldsymbol{b},\boldsymbol{Ab},\boldsymbol{A}^2\boldsymbol{b},\cdots,\boldsymbol{A}^{m-1}\boldsymbol{b})$;GMRES 通过求使残量 $\boldsymbol{Ax}_m-\boldsymbol{b}$ 最小的矢量 $\boldsymbol{x}_m\in\boldsymbol{K}_m$ 来逼近 $\boldsymbol{Ax}=\boldsymbol{b}$ 的精确解.但是,矢量 $\boldsymbol{b},\boldsymbol{Ab},\boldsymbol{A}^2\boldsymbol{b},\cdots,\boldsymbol{A}^{m-1}\boldsymbol{b}$ 几乎是线性相关的,因此通常采用 Arnoldi 迭代方法来找出正交矢量 $\boldsymbol{v}_1,\boldsymbol{v}_2,\cdots,\boldsymbol{v}_m$ 作为 m 阶 Krylov 子空间的基.故矢量 $\boldsymbol{x}_m\in\boldsymbol{K}_m$ 可写成 $\boldsymbol{x}_m=\boldsymbol{V}_m\boldsymbol{y}_m$,其中 $\boldsymbol{y}_m\in\mathbb{R}^m$ 且 $\boldsymbol{V}_m$ 是由 $\boldsymbol{v}_1,\boldsymbol{v}_2,\cdots,\boldsymbol{v}_m$ 组成的 $n\times m$ 矩阵.

通过 Arnoldi 迭代过程也可产生一个 $(m+1)\times m$ 阶的上 Hessenberg 矩阵 $\bar{\boldsymbol{H}}_n$ 满足 $\boldsymbol{AV}_m=\boldsymbol{V}_{m+1}\bar{\boldsymbol{H}}_m$.因为 $\boldsymbol{V}_m$ 是正交的,因此有 $\|\boldsymbol{Ax}_m-\boldsymbol{b}\|=\|\bar{\boldsymbol{H}}_m\boldsymbol{y}_m-\beta\boldsymbol{e}_1\|$;其中 $\boldsymbol{e}_1=(1,0,0,\cdots,0)$ 是 $\mathbb{R}^{m+1}$ 的标准基的第一个矢量,并且 $\beta=\|\boldsymbol{Ax}_0-\boldsymbol{b}\|$,$\boldsymbol{x}_0$ 是初始矢量(通常是零矢量).因此,求使得残量 $\boldsymbol{r}_m=\boldsymbol{Ax}_m-\boldsymbol{b}$ 范数最小的 $\boldsymbol{x}_m$,就变成求 $\boldsymbol{r}_m=\bar{\boldsymbol{H}}_m y_m-\beta\boldsymbol{e}_1$ 范数最小的问题,即为一个 m 阶线性最小二乘问题,通常情况下 $m\ll n$.

通过前面的描述可知,GMRES 算法在迭代的每一步中要进行如下操作:

Step1. 做一步 Arnoldi 迭代计算(见算法 1 中的第②~⑧步).

Step2. 寻找使得 $\|\boldsymbol{r}_m\|$ 最小的 $\boldsymbol{y}_m$(见算法 1 中的第⑨步).

Step3. 计算 $\boldsymbol{x}_m=\boldsymbol{x}_0+\boldsymbol{V}_m\boldsymbol{y}_m$.

Step4. 如果残量不够小,增大 Krylov 子空间维度并重复以上步骤.

本文采用的是重开始 GMRES 算法,即将 Step4 改为:如果残量不够小,将 $\boldsymbol{x}$ 的初值置为 $\boldsymbol{x}_m$ 并重复以上步骤.

算法 1. GMRES 算法[2]每次迭代的具体步骤

① 计算 $\boldsymbol{r}_0=\boldsymbol{b}-\boldsymbol{Ax}_0$,$\beta=\|\boldsymbol{r}_0\|$ 和 $\boldsymbol{v}_1=\boldsymbol{r}_0/\beta$;

② 初始化 $(m+1)\times m$ 阶的上 Hessenberg 矩阵 $\bar{\boldsymbol{H}}_n$ 为 0;

③ 循环 $j=1,2,\cdots,m$

④ 计算 $\boldsymbol{w}_j=\boldsymbol{Av}_j$;

⑤ 循环 $i=1,2,\cdots,j$

① http://en.wikipedia.org/wiki/Generalized_minimal_residual_method

⑥　计算 $h_{ij}=\boldsymbol{w}_j\cdot\boldsymbol{v}_i$；计算 $\boldsymbol{w}_j=\boldsymbol{w}_j-h_{ij}\boldsymbol{v}_i$

⑦　计算 $h_{j+1,j}=\|\boldsymbol{w}_j\|$，如果 $h_{j+1,j}=0$，则将 j 赋给 m，转到⑨；

⑧　计算 $\boldsymbol{v}_{j+1}=\boldsymbol{w}_j/h_{j+1,j}$

⑨ 计算 $\|\bar{\boldsymbol{H}}_m\boldsymbol{y}_m-\beta\boldsymbol{e}_1\|$ 的最小值，得到对应的 $\boldsymbol{y}_m$；

⑩ 计算 $\boldsymbol{x}_m=\boldsymbol{x}_0+\boldsymbol{V}_m\boldsymbol{y}_m$.

设系数矩阵 $\boldsymbol{A}$ 为 n 阶可逆矩阵，每行非零元数为 k，Krylov 子空间维度为 m（常数），则每次迭代中各种计算所需次数及复杂度如表 1 所示.

表 1　算法 1 中各计算步复杂度分析

计算步	复杂度	常系数	计算次数
稀疏矩阵与矢量乘积	$O(kn)$	2	$m+1$
矢量内积与范数	$O(n)$	2	$m(m+3)/2+1$
稠密矩阵与矢量乘积	$O(n)$	$2m$	1
矢量加减、矢量乘标量	$O(n)$	1	6
最小二乘求解	$O(1)$	$4m^2$	1

很显然，无论从计算量还是从 GPU 程序的优化难度来说，表 1 中的前 2 项都是整个算法中最耗时的部分.

设稀疏矩阵与矢量乘积、内积与范数的浮点运算次数分别为 f_1 和 f_2，则 $f_1=2nk(m+1)$，$f_2=nm(m+3)+2n$. 当 $k\gg\dfrac{m(m+3)+2}{2(m+1)}=\dfrac{m}{2}+1$ 时，稀疏矩阵与矢量乘积运算决定程序的整体性能；当 $k\ll\dfrac{m}{2}+1$ 时，矢量内积与范数运算决定程序的整体性能.

考虑到矢量内积和范数运算在 GPU 中运行比稀疏矩阵和矢量乘积遇到的性能瓶颈要少很多，因此可以认为 m 值越大，加速效果越好；k 值越大，加速效果越差.

尽管 m 较大时可以加速收敛和提高精度，但同时会使每次迭代的计算量增加并且使存储占用增加；另外由于浮点精度有限，因此 m 的值不宜取得过大. 至于 m 的值取多少合适，要考虑矩阵本身的特点、机器内存的限制、浮点精度的限制等，一般根据矩阵条件数凭经验决定. 另外，过小的 m 可能导致收敛停滞，文献[4]对此进行了一些讨论.

2　GMRES 算法的 CUDA 优化实现

关于 CUDA 编程的详细说明可以参考 NVIDIA 手册[7,12]，本文不再赘述. 根据本文前面的描述，GMRES 算法跟其他线性方程组迭代法求解类似. 整个计算被分解成稀疏矩阵与矢量乘积运算，矢量内积与范数，稠密矩阵与矢量乘积，矢量加减、矢量乘标量，最小二乘求解等几步. 为了获得整体优化的效果，本文对于不同的计算步骤采用不同的策略.

2.1　稀疏矩阵与矢量乘积运算

本文中矩阵存储使用压缩稀疏行存储（compressed sparse row，CSR）格式，即将所有非零元素存入一个数组 *data*，这些非零元素的列号存入另外一个数组 *indices*，而数组 *rpos* 存放每一行第一个非零元素在数组 *data* 中的序号，*rpos* 中的最后一个元素存放总的非零元素的个数. CSR 格式稀疏矩阵与矢量乘积在 GPU 中计算遇到的瓶颈以及一些算法实现见文献[11]. 本文改进了翟艳堂等的稀疏矩阵与矢量乘积运算方法①，对矩阵数据读取全部实现了合并访问，取得了很好的加速效果.

乘法与加法分为两步的思想较为简单，其步骤如下：

Step1. Kernel1. 计算每个非零元与相应标量的乘积，保存结果至全局存储器，用 *mres* 表示.

Step2. Kernel2. 读取 Kernel1 的结果，并将每行对应的数据累加得到结果矢量.

Kernel1 较为简单，这里不再赘述，详细请参考相关链接. 对于 Kernel2，使用纹理存储并不能解决非合并数据读取大量访问延时造成的性能瓶颈.

本文提出合并访问的 Kernel2 算法（以 *block* 大小为 128 为例）.

算法 2. 稀疏矩阵与矢量乘积运算第 2 步核心

Step1. *tid_b*←*thread* index in Block

Step2. *tid_g*←*thread* index in Grid

Step3. *sum*←0.0

Step4. define shared array *rpos_s*[129] and *mres_s*[128]

Step5. *rpos_s*[*tid_b*]←*rpos*[*tid_g*]

Step6. if *tid_b*=0 then *rpos_s*[129]←*rpos*[*tid_g*+128]

Step7. *baseAddress*←*rpos_s*[0]

Step8. *mres_s*[*tid_b*]←*mres*[*baseAddress*+*tid_b*]

Step9. $P\leftarrow\{x\mid rpos\_s[tid\_b]\leqslant x<rpos\_s[tid\_b+1]\}\cap\{x\mid baseAddress\leqslant x<baseAddress+128\}$

Step10. if $P=\varnothing$ goto Step13

① http://cuda.csdn.net/Contest/pro/nvidia_showme.aspx?pointid=52

Step11. for all elements i in P

Step12. $sum+=mres\_s[i]$

Step13. $baseAddress \leftarrow bassAddress+128$

Step14. if $baseAddress<rpos\_s[129]$ goto Step8

Step15. $result[tid\_g] \leftarrow sum$.

其中，$rpos\_s$ 为每个 block 分配的共享内存(shared memory)，用于存放该 block 所处理的某些行第一个非零元素在数组 $data$ 中的序号；$mres\_s$ 也是每个 block 分配的共享内存，用于存放该 block 所处理的非零元与相应标量的乘积. 通过共享内存，可以大大加速数据访问，减少延迟.

本文通过分析发现上述算法存在如下不足：假设稀疏矩阵平均每行非零元素为 k，则每个 block 中同时执行 Step12 的线程约为 $\frac{128}{k}$，对于 $k=10$，实际工作的只有 13 个线程，远不足一个 warp(一个 warp 由 32 个线程组成). 也就是说从 Step8～Step14 的每一次迭代中，当程序执行到 Step12 时，Streaming Multiprocessors (SM)的利用率只有 $\frac{13}{32}$. 更糟糕的是，Step12 是上述算法计算量最多的一步.

解决的办法是扩大 $mres\_s$ 的容量. 例如 $mres\_s$ 的长度为 512，对于 $k=10$，同时执行 Step12 的线程数约为 51 个. 假设第一个线程在 block 中的编号为 tid，则 $tid\%32$ 为其在 warp 块中的编号. 当 $tid\%32 \leqslant 12$ 时，这 51 个线程占用 2 个相邻 warp 块；当 $tid\%32 > 12$ 时，这 51 个线程占用 3 个相邻 warp 块. 因此，Step12 中 SM 的利用率期望值为

$$\frac{13}{32}\cdot\frac{51}{64}+\frac{19}{32}\cdot\frac{51}{96}\approx 0.64.$$

类似地可以计算当 $k=5$ 时，SM 的利用率约为 0.77.

另外，扩大 $mres\_s$ 的容量可以减少从 Step8～Step14 的迭代次数，因此该算法效果很好. 例如当 $mres\_s$ 容量为 512，$k<4$ 时，从 Step8～Step14 的迭代在大多数情况下只需要进行一次. 也就是说，所有需要的数据被一次性读入共享内存，然后所有线程同时对共享内存中的数据进行操作.

对 Step9 求交集，可以根据 $\{x|a\leqslant x<b\}\cap\{x|c\leqslant x<d\}\neq\varnothing\Leftrightarrow\neg(b\leqslant c\vee a\geqslant d)$ 很容易地完成.

本文与文献[11]中的稀疏矩阵和矢量乘积算法进行了代码级性能比较，结果如表 2 所示，其中本文所用的矩阵数据全部来自 http://www.cise.ufl.edu/research/sparse/matrices.

表 2 稀疏矩阵和矢量乘积算法效率比较

矩阵名称	矩阵规模	平均每行非零元个数	计算 100 次耗时/s		
			本文算法	文献[11]中算法 1	文献[11]中算法 2
c-26	4307×4307	4.5	0.006	0.039	0.005
crystk02	13965×13965	35.2	0.017	0.047	0.020
Language	399130×399130	3.0	0.193	1.224	0.369
Largebasis	440020×440020	12.6	0.163	0.569	0.325
ASIC_680ks	682712×682712	3.4	0.104	0.380	运行失败
G3_circuit	1585478×1585478	2.9	0.145	0.193	运行失败
kkt_power	2063494×2063494	3.9	0.351	0.698	运行失败
memchip	2707524×2707524	5.5	0.446	1.001	运行失败
Freescale1	3428755×3428755	5.5	0.600	1.245	运行失败

由表 2 可以看出，本文实现的稀疏矩阵和矢量乘积算法要比 NVIDIA 提供的算法效率高，而且对于一些矩阵，NVIDIA 提供的算法存在失效的情况.

2.2 矢量内积和范数运算

算法中矢量范数采用欧几里德范数，由于它与矢量内积计算类似，因此本文只介绍矢量内积的计算.

内积计算的困难在于如何在保持一定并行度的情况下将每个标量乘积累加起来. 本文采用 reduction[①] 的思想，即将 2 个输入矢量划分成若干对小矢量，每个 block 负责计算一对小矢量的内积，这些小矢量的内积被写到 mapped memory 中，由 CPU 负责将这些小矢量内积加起来得到最终结果.

① http://en.wikipedia.org/wiki/Reduction_(complexity)

为了对不同问题规模都能得到最佳的 warp 块装载量，本文并未固定小矢量的长度，而是在限制 block 个数不大于 512 的情况下动态地决定小矢量的长度，因此得到的小矢量不多于 512 对.

设输入矢量长度为 n，block 大小为 b，block 个数为 g，小矢量长度为 m，则有如图 1 所示的判定树.

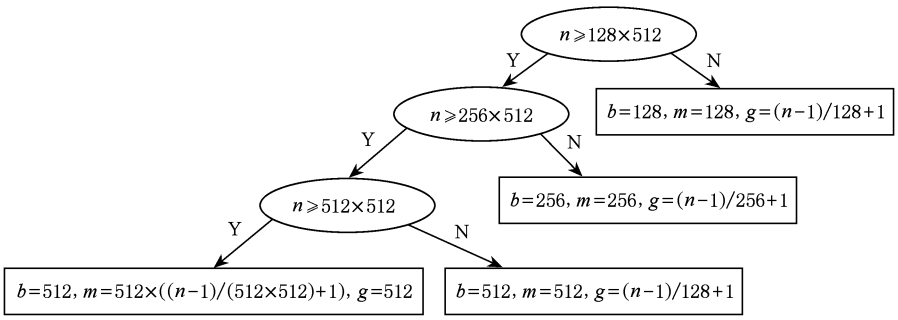


图 1　矢量内积计算判定树

从图 1 中可以看出，利用该判定树得到的线程和数据划分在任何时候都不会对 warp 块装载量产生影响. 当 $n \geqslant 512 \times 512$ 时，尽管此时 block 的数量被限制为 512，但由于每个 SM 只能同时装载 2 个大小为 512 的 block，所以这个限制是合理的，不会对性能产生影响.

以 block 大小为 128 为例（图 1 判定树根的右节点），kernel 函数代码如下：

算法 3. CUDA 内积运算核心

```
_global_void innerPro_128(float * v, float * w, float * mres){
  int tid=threadIdx.x;
  int tid_in_grid = blockIdx.x * blockDim.x + threadIdx.x;
  _shared_ float r_s[128];
  r_s[tid]=v[tid_in_grid] * w[tid_in_grid];
  _syncthreads();
  if (tid < 64) r_s[tid] += r_s[tid + 64];
  _syncthreads();
  if(tid<32) {
    r_s[tid]+= r_s[tid+32];
    if(tid<16) r_s[tid]+= r_s[tid+16];
    if(tid<8) r_s[tid]+= r_s[tid+8 ];
    if(tid<4) r_s[tid]+= r_s[tid+4 ];
    if(tid<2) r_s[tid]+= r_s[tid+2 ];
    if(tid<1) mres[ blockIdx.x ]=r_s[0]+r_s[1];
  }
}
```

其中 $\boldsymbol{v}$ 和 $\boldsymbol{w}$ 分别为输入的 2 个矢量，而 ***mres*** 则为生成的小矢量.

矢量内积计算遇到的另一个问题是数组求和的误差舍入. 例如，有一个长度为 1 000 000 的数组，每个元素均为 0.001，使用 IEEE 标准单精度顺序相加得到的结果是 991.141 541，和正确结果 1 000 相差接近百分之一. 这是由于浮点精度导致的舍入误差造成的. 使用 reduction 方法的情况要好很多，如算法 3 代码所示，数组元素是两两相加的. 当数组元素的数量级差别不大时，使用 reduction 方法得到的结果几乎是精确的；但当数组元素数量级相差较大时，其结果是偏小的，如 1.0E10+1.0E−10=1.0E10. 解决加法误差的方法有 Kahan summation 算法①，但其会造成性能上的大幅度下降，因此本文并未采用.

2.3　稠密矩阵矢量乘积、矢量加减、矢量乘标量和最小二乘求解

稠密矩阵矢量乘积、矢量加减和矢量乘标量问题较为简单，本文采用 CUDA 实现. CUDA SDK 中提供了类似的例子代码，这里不在赘述. 求解最小二乘问题 $\boldsymbol{y}=\arg\min \| \bar{\boldsymbol{H}}_m \boldsymbol{y} - \beta \boldsymbol{e}_1 \|_2$ 较为烦琐[1-2]，但从表 1 中可以看出，最小二乘问题的浮点运算次数大约为 $4m^2$，计算量并不大，没有构成问题的瓶颈，因此该计算在 CPU 端完成.

3　实验与结果讨论

本文算法在 Visual Studio 2005 + Window XP/Vista 软件环境下，采用 C++语言结合 CUDA 3.1 编程实现. GPU 为 NVIDIA GeForce GTX260，分别在 2 台装有不同 CPU 的机器上运行，机器 1 为

① http://en.wikipedia.org/wiki/Kahan_summation_algorithm

Intel Core 2 Quad CPU Q9400@2.66 GHz,得到了平均 40 余倍的加速效果,机器 2 为 Intel Core i7 CPU 920 @2.67 GHz,得到平均 20 余倍的加速效果.它们均迭代 10 次,具体实验结果如表 3 所示.

表 3 机器 1,2 上的性能测试

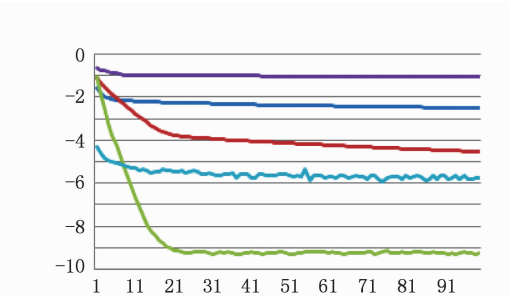
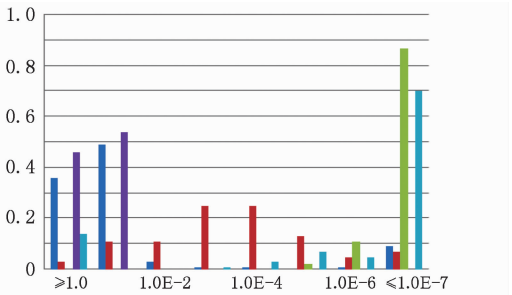
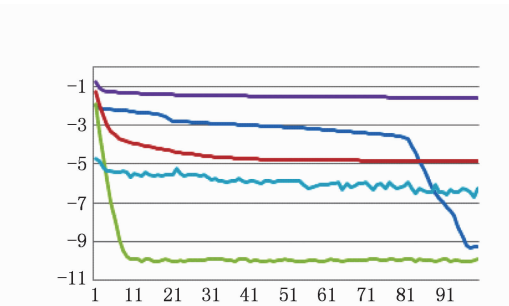
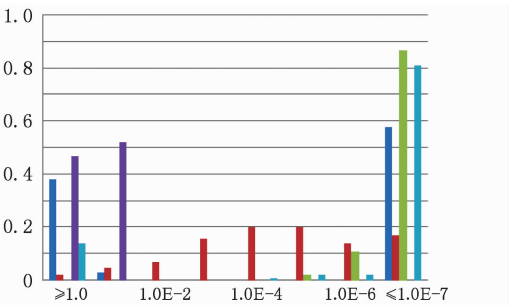
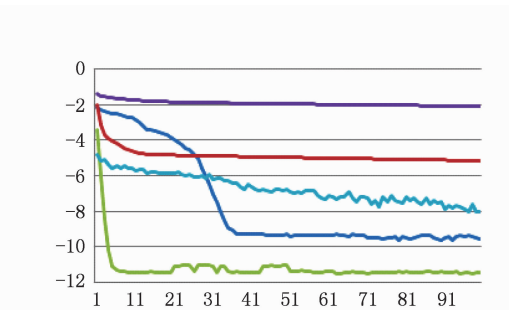
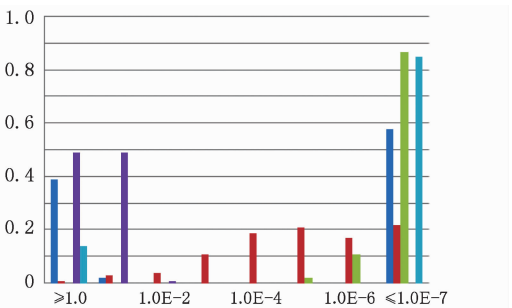
机器 1				机器 2(以灰色底纹区分)					
矩阵名称	矩阵规模	平均每行非零元数	Krylov 维度	CPU 时间/s		GPU 时间/s		加速比	
c-26	4307×4307	4.5	8	0.015		0.016		0.9	
			16	0.047		0.063		0.7	
			32	0.140		0.172		0.8	
crystk02	13965×13965	35.2	8	0.187		0.031		6.0	
			16	0.343		0.063		5.4	
			32	0.390		0.094		4.1	
Language	399130×399130	3.0	8	7.594		0.234		32.5	
			16	52.422		0.563		93.1	
			32	239.203		1.484		161.2	
Largebasis	440020×440020	12.6	8	4.047	2.610	0.218	0.350	18.6	10.4
			16	10.312	6.624	0.500	0.641	20.6	10.3
			32	34.562	19.593	1.406	1.934	24.6	10.1
ASIC_680ks	682712×682712	3.4	8	5.250	2.847	0.187	0.247	28.1	11.5
			16	14.954	8.638	0.484	0.601	30.9	14.4
			32	51.235	28.458	1.531	1.860	33.5	15.3
G3_circuit	1585478×1585478	2.9	8	11.046	6.938	0.281	0.309	39.3	22.5
			16	33.109	20.967	0.765	0.877	43.3	23.9
			32	110.797	69.005	2.532	2.681	43.8	25.7
kkt_power	2063494×2063494	3.9	8	16.375	9.870	0.532	0.543	30.8	18.2
			16	45.234	29.160	1.297	1.367	34.9	21.3
			32	157.063	102.412	5.266	4.043	31.8	25.3
memchip	2707524×2707524	5.5	8	21.407		0.672		31.9	
			16	61.782		3.062		20.2	
			32	208.031		4.782		43.5	
Freescale1	3428755×3428755	5.5	8	27.234		0.875		31.1	
			16	76.688		2.187		35.1	
			32	258.297		6.344		40.7	

由于机器 2 的 CPU 配置比机器 1 高 1 倍,导致机器 2 上的 GMRES 算法的 CPU 版本进行的效率比机器 1 的提升了 2 倍,但同样的 GPU 版本代码效率却略有下降,这可能是数据总线传输速率差异导致的.

为了确保 GPU 与 CPU 运算结果一致,本文进行了收敛曲线和误差分布分析,总共迭代 100 次,取了 5 个矩阵样本;其中测试矩阵得到的收敛曲线和误差分步直方图如表 4 所示.

从表 4 可以看出,曲线收敛速度跟矩阵关系比较紧密,有些收敛得快,有些收敛得慢,说明收敛效果首先取决于方程组系数矩阵的实际情况.对于同一个矩阵来说,Krylov 子空间维度的大小则对于收敛有着至关重要的影响.另外可以看出,rajat26 和 c-26 2 个矩阵的求解并没有收敛到近似解,这是矩阵的条件数较大对误差较为敏感造成的,从收敛曲线也可看到这 2 个矩阵的计算收敛情况出现异常.

表 4　收敛曲线与误差分步直方图

子空间维度	收敛曲线（横坐标为迭代次数，纵坐标为 $\lg\frac{\|\boldsymbol{Ax}-\boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2}$）	误差分步直方图（横坐标为标量误差 $\|x_i'-x_i\|/\|x_i\|$，纵坐标为所占比例）
8		
16		
32		

4　结论和未来工作

本文通过对 GMRES 算法进行算法分析，并利用 CUDA 实现线性方程组的快速求解. 尤其是对于稀疏矩阵与矢量乘积运算部分，通过合并访问和共享内存策略提高数据负载的利用率，使得对于大规模线性方程组的求解效率得到提升，对于当前主流的 PC 机来说，GPU 版本要比 CPU 版本快 20～40 倍，对于某些矩阵效率提升更多. 由于 GMRES 迭代算法适用于一般的线性方程组的求解，所以只要求系数矩阵可逆就有解. 虽然条件数对迭代收敛结果会有影响，相信在工程应用领域会大有用武之地. 我们拟将本文中的求解算法应用到流体仿真和变形计算的计算机动画研究中去.

致谢　感谢 NVIDIA 公司提供 GTX260 显卡，感谢佛罗里达大学提供大规模稀疏矩阵测试数据！

参考文献(References)：

[1] Saad Y, Schultz M H. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems [J]. SIAM Journal on Scientific and Statistical Computing, 1986, 7(3): 856-869

[2] Saad Y. Iterative methods for sparse linear systems [M]. 2nd ed. Philadelphia: SIAM, 2003

[3] Quan Zhong, Xiang Shuhuang. A GMRES based polynomial preconditioning algorithm [J]. Mathematica Numerica Sinica, 2006, 28(4): 365-376 (in Chinese)
(全 忠, 向淑晃. 基于 GMRES 的多项式预处理广义极小残差法[J]. 计算数学, 2006, 28(4): 365-376)

[4] Habu M, Nodera T. GMRES(m) algorithm with changing the restart cycle adaptively [C] //Proceedings of Algoritmy Conference on Scientific Computing. Heidelberg: Springer, 2000: 254-263

[5] Wu Enhua, Liu Youquan. General purpose computation on GPU [J]. Journal of Computer-Aided Design & Computer Graphics, 2004, 16(5): 601-612 (in Chinese)
(吴恩华, 柳有权. 基于图形处理器(GPU)的通用计算[J]. 计算机辅助设计与图形学学报, 2004, 16(5): 601-612)

[6] Wu E H, Liu Y Q. Emerging technology about GPGPU [C] //Proceedings of IEEE Asia Pacific Conference on Circuits and Systems. Los Alamitos: IEEE Computer Society Press, 2008: 618-622

[7] NVIDIA CUDA C programming guide. Version 3.1 [M]. San Jose: NVIDIA, 2010

[8] Wang M L, Klie H, Parashar M, *et al*. Solving sparse linear systems on NVIDIA tesla GPUs [M] //Lecture Notes in Computer Science. Heidelberg: Springer, 2009, 5544: 864-873

[9] Velamparambil S, MacKinnon-Cormier S, Perry J, *et al*. GPU accelerated Krylov subspace methods for computational electromagnetics [C] //Proceedings of the 38th European Microwave Conference. Los Alamitos: IEEE Computer Society Press, 2008: 1312-1314

[10] Ghaemian N, Abdollahzadeh A, Heinemann Z, *et al*. Accelerating the GMRES iterative linear solver of an oil reservoir simulator using the multi-processing power of compute unified device architecture of graphics cards [C] //Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing. Heidelberg: Springer, 2008: 156-159

[11] Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA [R]. San Jose: NVIDIA, NVR-2008-004, 2008

[12] NVIDIA CUDA C best practices guide. Version 3.1 [M]. San Jose: NVIDIA, 2010