代数多重网格在 GPU 上的优化研究

李佳佳[1+]，张秀霞[1]，谭光明[1]，陈明宇[1]

[1](中国科学院计算技术研究所，北 京 100190)

Algebraic Multi-grid optimization study on GPU

LI Jia-Jia[1+], ZHANG Xiu-Xia[1], TAN Guang-Ming[1], CHEN Ming-Yu[1]

[1](Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: Phn +86-010-62600670, Fax +86-010-62600600, E-mail: lijiajia@ict.ac.cn, http://http://asl.ncic.ac.cn/ljj/

Received 2011-07-30; Accepted 2011-09-15

Abstract: Recently, as applied widely in scientific computing, GPU plays a more and more important role in high performance computing. The third generation of NVIDIA architecture – Fermi significantly improves floating point computation performance in double-precision. It is a necessity to optimize Algebraic Multigrid (AMG) on GPU, which is an important iterative algorithm in high performance computing area. In this paper, we optimize AMG solution of sparse linear equations on GPU, and AMG-GPU performance has 2X improvement compared with implemented on CPU. However, limited by error accuracy of 1e-6, the performance improvement is merely 6% due to the smaller number of cycles. We analyze AMG-GPU in detail, and draw a conclusion that the major factor influencing AMG-GPU performance is the dynamic changes of operators scale at each level.

Key words: High Performance Computing; Algebraic Multigrid; GPU; CUDA; SpMV

摘 要: 近年来，随着 GPU 更多地用于科学计算，在高性能计算领域起着越来越重要的作用。NVIDIA 第三代 Fermi 架构的开发，大大提升了 GPU 的双精度浮点运算能力。代数多重网格(AMG)作为高性能计算领域中的重要迭代算法，利用 GPU 的高浮点计算能力对其优化是很有必要的。本文基于求解稀疏代数方程组对 AMG 算法进行GPU优化，AMG-GPU的性能提升为原始AMG的2倍。但在1e-6的精度要求下，由于迭代次数的减少AMG-GPU的性能仅提升约6%。我们对AMG-GPU进行了详细剖析，得出AMG各层操作算子规模的动态变化是影响AMG性能提升的关键因素。

关键词: 高性能计算; 代数多重网格; GPU; CUDA; SpMV

中图法分类号: TP302 **文献标识码**: A

1 引言

近年来，图形处理器(GPU)由于其高浮点峰值性能，越来越多地应用于高性能计算领域。NVIDIA Tesla C2070 [1]的单精度浮点峰值性能超过 1TFLOP/s，Fermi 体系结构的出现使得 GPU 双精度浮点性能大幅度提升，如 Tesla C2070 的双精度浮点峰值性能达到 515 GFLOP/s，这进一步促进了 GPU 在高性能领域中的广泛使用。代数多重网格（简称“AMG”）[2,3]，是高性能领域求解稀疏线性方程组的重要方法，广泛应用于地

* Supported by National 863 Program under Grant No.2009AA01A129（国家 863 计划）; the National Natural Science Foundation of China under Grant No.60803030,61033009,60921002,60925009（国家自然科学基金）; the 973 Program under Grant No.2011CB302500（国家 973 计划）.

作者简介：李佳佳(1988－),女,辽宁省北票市人,博士研究生,学生,主要研究领域为并行计算;张秀霞(1987－),女,硕士研究生,学生,主要研究领域为并行计算;谭光明(1980－),男,博士,副研究员,主要研究领域为并行计算;陈明宇(1972－),男,博士,研究员,主要研究领域为计算机体系结构.

下水勘探、爆炸性材料建模、电磁学应用、核物理应用等领域。LLNL 开发了大规模集群上 AMG 的并行算法 BoomerAMG[4]，集成在 hypre 代数库[5]中。多重网格方法具有很好的算法可扩展性，即求解每个非零元所需的操作数可以为常量，不随并行规模的扩大而增加。这一性质决定了 BoomerAMG 在大规模集群上具有良好的扩展性。当前，曙光和国防科技大学都研制出浮点计算能力达到 P 级的超级计算机（Tianhe-1A 和 Nebulae）[6]，两者均使用 GPU 作为加速器件。因此，在 GPU 上对高性能计算进行优化是未来发展的趋势。近来 NVIDIA 推出第三代 Fermi 架构，其改进之一是双精度浮点型峰值性能的提升，这使得 GPU 更适合对科学计算进行优化，如 AMG。

2 AMG 简介

2.1 AMG概述

AMG 不需要进行具体的几何网格划分，而是根据线性方程组待求向量构建虚拟网格层。虚拟网格层上的操作算子及插值和粗化算子的构建都基于线性方程组的系数矩阵，AMG 的具体实现细节请见[2,3]。使用 AMG 算法求解稀疏线性方程组 Au=f（其中 A 为$n \times n$稀疏矩阵，u, f 为稠密向量，已知向量 f，求解向量 u），包含两个阶段：建立阶段和求解阶段。在建立阶段，根据已知系数矩阵 A 构造虚拟的网格层$\Omega^1, \Omega^2, \cdots, \Omega^M$（其中 M 为最大网格层数），在各网格层上构建操作算子$A^1, A^2, \cdots, A^M$，相邻网格层间的转换算子——插值算子I_{h+1}^h, $h = 1,2, \cdots, M-1$和限制算子I_h^{h+1}, $h = 1,2, \cdots, M-1$。在求解阶段，根据第一阶段构造的各网格层和各层算子，用多重网格法求解。首先对$1,2, \cdots, M-1$网格层用光滑算法迭代μ次，求出余差r^m $(m = 1,2, \cdots, M-1)$并对其进行粗化，继而传递到下一层粗网格并继续刚才的过程。在最粗的网格层，用 Gauss 消元法直接求解$A^M e^M = r^M$。每层求解出的误差值e^M进行插值并传递到上一层细网格，最后用光滑算法迭代μ次。上述求解的全过程见图 1 和图 2。本文我们关注 AMG 的求解过程，可以看出求解过程的主要组成部分是稀疏矩阵向量乘法（SpMV）和光滑操作。AMG 中的光滑操作通常使用 Jacobi 迭代、Gauss-Seidel 迭代、SOR 迭代等，其中多次调用类似于 SpMV 的操作（简称为“类 SpMV”）。因此可以认为求解过程中的主要操作为 SpMV 和类 SpMV，经测试两者占 AMG 求解过程的 90+%。由于光滑算法中 Jacobi 迭代具有最好的可并行性，以下我们以 Jacobi 迭代为例讨论。

AMG 的建立阶段：

(1) 设置 h=1

(2) 将Ω^k分为不相交的两个子集C^h和F^h

  a) 设置$\Omega^{h+1} = C^h$

  b) 定义插值算子I_{h+1}^h

(3) 设置$I_{h+1}^h = \left(I_{h+1}^h\right)^T$，求出每层的操作算子$A^{h+1} = I_h^{h+1} A^h I_{h+1}^h$

(4) 当Ω^{h+1}足够小时，设置 M=h+1，结束。否则，置 h=h+1，并转至(2)

AMG 求解阶段： $MV^h(A^h, u^h, f^h, \mu)$

1. 若 h=M，设置$u^M = (A^M)^{-1} f^M$
2. 否则:
   a) 前光滑：对$A^h x^h = b^h$作 μ 次简单迭代：$e^h \leftarrow S^\mu e^h$
   b) 在粗网格Ω^h上求解：
      i. 残差计算和限制：$r^H = R(b^h - A^h x^h)$;
      ii. 求解：$A^H e^H = r^H$;
   c) 插值并校正原近似解：$x^h \leftarrow x^h + P e^H$;
   d) 后光滑：对$A^h x^h = b^h$作μ次简单迭代：$e^h \leftarrow S^\mu e^h$;

图 1 AMG 算法

2.2 AMG中SpMV的实现

BoomerAMG 中使用 CSR（Compressed Sparse Row）格式存储系数矩阵及相关操作算子。CSR 格式是稀疏矩阵中最常用的存储格式，是一种通用存储模式。该存储模式只记录稀疏矩阵中的非零元素，用三个数组分别存储矩阵中的非零元素 data，每个非零元对应的列下标 indices 和每行非零元的起始位置 ptx，如图 3。CSR 格式对应的 SpMV 实现见图 4。

3 AMG 在 GPU 上的优化

3.1 GPU概述

GPU 具有线程数目庞大、浮点峰值性能高和带宽高的特点，具有很高的并行度。NVIDIA 新一代 Fermi 架构一个重要的改进是提高了双精度浮点性能，如 Tesla C2070 双精度浮点峰值性能为 515GFLOP/s，这是主流 CPU 的数倍。GPU 的高浮点计算能力吸引了大量应用在其上进行优化，有些应用取得了良好的性能提升。本文实现了 AMG 在 GPU 上的优化，并对实验结果进行了详细地分析。

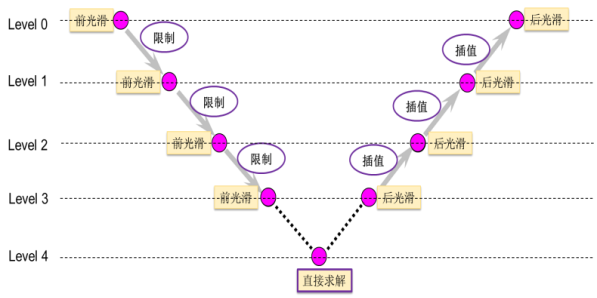

图 2 AMG 算法中求解过程执行情况

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

ptr = [0 2 4 7 9]

indices = [0 1 1 2 0 2 3 1 3]

data = [1 7 2 8 5 3 9 6 4]

图 3 CSR 格式存储矩阵的一个例子

3.2 AMG-GPU实现

SpMV GPU 实现 在 2.3 中，我们得出 AMG 的求解过程中，主要操作为 SpMV 和 Jacobi 迭代操作，因此我们针对这两种操作在 GPU 上进行优化。SpMV 对稀疏矩阵的一行进行操作，可以看成是两个向量的点乘，计算结果存在向量 y 的对应元素中。由于每行的点乘操作之间没有依赖关系，可以独立执行。因此我们作 GPU 优化时，每个线程执行原矩阵中一行的点乘操作，使各线程间充分并行，GPU kernel 部分代码见图 5。对于 Jacobi 迭代，我们对其中的类 SpMV 操作进行 GPU 优化，如图 6。类 SpMV 与 SpMV 的区别在于，前者比后者多一个对矩阵中的一行是否进行点乘操作进行的条件判断，我们的优化方法类似于 SpMV，如图 7。

```
void
spmv_csr (int num_rows, int *ptr, int *indicies,
double *data, double *x, double *y)
{
   for (int i=0; i<num_rows; i++)
   {
      double dot = 0;
      int row_start = ptr[i];
      int row_end = ptr[i+1];
      for (int jj=row_start; jj<row_end; jj++)
         dot +=data[jj]*x[indices[jj]];
      y[i] += dot;
   }
}
```

图 4 CSR 格式 SpMV 算法实现

```
__global__ void
spmv_csr_kernel (int num_rows, int *ptr, int
*indicies, double *data, double *x, double *y)
{
   int row = blockdim.x*blockdim.x+threadidx.x;
   if (row < num_rows)
   {
      double dot = 0;
      int row_start = ptr[row];
      int row_end = ptr[row+1];
      for (int jj=row_start; jj<row_end; jj++)
        dot +=data[jj]*x[indices[jj]];
      y[row] += dot;
   }
}
```

图 5 CSR-SpMV 的 GPU 实现

SpMV kernel 线程分配　GPU 上线程的划分有三级，gird, block 和 thread，每一级划分大小在不同的 GPU 平台上限制也不同。并且不同的逻辑划分方式会对线程间的共享数据的使用产生影响。同一 block 内部的 threads 间可以进行同步，而不同 blocks 之间不能同步。SpMV 的 GPU kernel 中，由于矩阵规模的不定性，单纯使用 thread 级并行是不可行的。因此，我们根据矩阵规模动态划分 block 和 thread 的大小，形成两层的逻辑结构，每个 SpMV kernel 对应一个 grid。

AMG-GPU 的数据传输　至此，我们把求解过程的两个关键操作：SpMV 和 Jacobi 迭代都移植到 GPU 上执行，接下来考虑 CPU 与 GPU 间的数据传输问题。由于 GPU 需要计算的原始数据都存储在 CPU 的内存中，而 GPU 只有当数据存在于显存上时才能进行计算，这就产生了数据传输问题。我们在 AMG 的建立过程后，集中对所有求解层次的算子从 CPU 到 GPU 进行数据传输，其中包括操作算子$A^1, A^2, \cdots, A^M$，相邻网格层间的转换算子——插值算子I_{h+1}^{h}, $h = 1,2,\cdots,M-1$和限制算子I_{h}^{h+1}, $h = 1,2,\cdots,M-1$，这样所有 GPU 所需的数据均提前存储在 GPU 显存中。这部分时间不计入求解时间。在每个 cycle 执行完成后，都生成一个向量 u。这个向量 u 是否满精度要求，需要在 CPU 上求范数来决定。因此在每个 cycle 中的向量 u 都需要从 GPU 传输到 CPU，这个过程在求解内部进行，因此这部分的传输时间所占比例不能过大。

```
void
jacobi_csr (int num_rows, int relax_points, int
relax_weight, int *cf_marker, int *ptr, int *indicies,
double *data, double *Vtemp, double *f, double *u)
{
  for (int i=0; i<num_rows; i++)
  {
    if (cf_marker[i] == relax_points &&
data[ptx[i]] != zero)
    {
    double res = f[i];
    for (int jj=ptx[i]+1; jj<ptx[i+1]; jj++)
      res +=data[jj]*Vtemp[indices[jj]];
    u[i] *= (1-relax_weight);
    u[i] += relax_weight * res / data[ptx[i]];
  }
}
```

图 6　Jacobi 迭代中类 SpMV 在 CPU 上的实现

```
__global__ void
jacobi_csr_kernel (int num_rows, int relax_points,
int relax_weight, int *cf_marker, int *ptr, int
*indicies, double *data, double *Vtemp, double *f,
double *u)
{
  int row = blockdim.x*blockdim.x+threadidx.x;
  if (row < num_rows)
  {
    if (cf_marker[row] == relax_points &&
data[ptx[row]] != zero)
    {
    double res = f[row];
    for (int jj=ptx[row]+1; jj<ptx[row+1]; jj++)
      res +=data[jj]*Vtemp[indices[jj]];
    u[row] *= (1-relax_weight);
    u[row] += relax_weight * res / data[ptx[row]];
  }
}
```

图 7　Jacobi 迭代中类 SpMV 在 GPU 的 kernel

4　实验结果及分析

4.1　实验结果

我们在两个平台上测试了 AMG-GPU 性能，平台 1 的 CPU：Intel X5650，GPU：Tesla C2050，使用 CUDA_SDK3.0。平台 2 的 CPU 为 Intel Core7，GPU 为 GeForce GTX480，使用 CUDA_SDK3.0。本文关注结构化矩阵，因此实验数据选为 1Million*1Million 的五对角矩阵，非零元总数为 5Million。AMG 的建立阶段使用 Falgout 粗化方法，求解过程采用 V-cycle，光滑算法使用 Jacobi 迭代法。两个平台的实验结果分别见表 1 和图 6。

从表 1 中可知当运行 200 次 V-cycle 时，AMG-GPU 求解过程的时间为为单进程 CPU 的一半，即性能是单进程 CPU 的 2 倍，与 AMG 在 4 个进程的 CPU 性能相当。当容忍误差设定为 1e-6 时（如图 6），由于只需

运行 8 个 V-cycle 即可达到精度需求，AMG-GPU 的性能仅比串行 CPU 版 AMG 高 6%，相比 4 个 OpenMP 线程或 4 个 MPI 进程的 CPU 版 AMG 性能分别降低 42%和 29%。

进程个数	1	4
CPU	58.4s	27.2s
GPU	28s	-
加速比	2.08	0.97

表 1 平台 1 上 AMG-GPU 中求解过程的执行时间与 CPU 的对比，运行了 200 个 cycles。

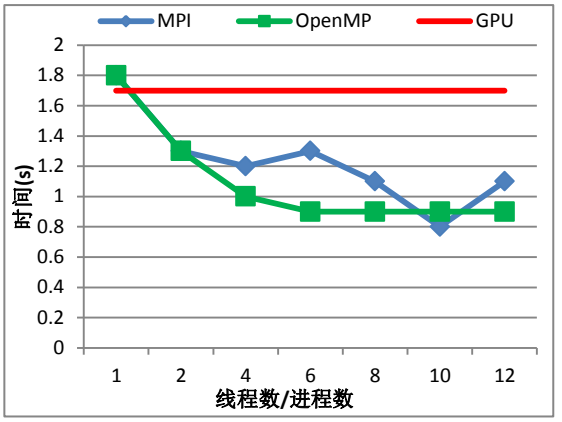

图 8 平台 2 上 AMG-GPU 求解过程的执行时间与 CPU 上多进程/线程的对比，误差精度范围是 1e-6。

	CPU		GPU	
Level	Relax Average	SpMV kernel	Relax Average	SpMV kernel
0	1286	768	2933	2921
1	1902	923	3094	1548
2	1446	1056	1325	826
3	2129	1083	1505	769
4	1596	1121	1324	688
5	2274	1210	1822	638
6	2239	1568	1709	642
7	2085	1544	851	486
8	1811	1351	232	141
9	1647	1275	146	102
10	1584	1076	66	57
11	1701	788	24	24
12	1011	232	7	8

表 2 各层 SpMV 和 Jacobi 迭代的浮点性能，单位为 MFLOP/s

时间	建立阶段	求解阶段	数据初始化
sec	2.24	1.78	2.39

表 3 AMG-GPU 各阶段时间统计

时间	solve 过程	Relax	粗化	插值	数据传输
ms	189	104	48	2	8

表 4 AMG-GPU 求解过程中 1 次 V-cycle 的操作时间分布

4.2 实验分析

4.2.1 SpMV 效率分析

我们测试得到每层 SpMV 和 Jacobi 迭代的浮点性能，如表 2。从图中可知，在 Level 0（即最细网格层）其浮点性能可达到 3GFLOP/s。这与 Bell 等人在[7]中的实验结果一致，说明我们的 SpMV 及类 SpMV 优化达到目前最优水平。在 Level 0 和 1 中，SpMV 和 Relax 相对 CPU 均有 2~3 倍的加速比。但随着层数的增加，

矩阵规模的降低，GPU 在 SpMV 和 Relax 上的优势减弱甚至性能低于 CPU。由于 GPU 需要大量的线程来掩盖访存延迟，当矩阵规模减小时，GPU 的线程数不足以掩盖访存延迟的开销，导致 kernel 性能急剧下降。而 CPU 不存在这一特性，因此 CPU SpMV 性能随矩阵规模变化波动较小。

4.2.2 数据传输分析

AMG-GPU 中含有两部分数据传输：一是建立过程之后，将各层操作算子、插值算子、粗化算子和初始向量 f 及 u 传输到 GPU 显存；二是求解过程中，每层 GPU 计算得到的向量 u 需要传输回 CPU 计算范数及相关处理。如表 3，第一次数据传输即数据初始化过程，虽然只进行一次，但由于传输数据量大，仍占用了整个 AMG 时间 37%。表 4 给出了求解过程中一次 V-cycle 的主要操作的时间分布，其中 Relax 占时间最多，为 55%；其次是粗化和插值，即 SpMV 过程，占 26%；数据传输仅占 4%。因此，可以认为求解过程中的数据传输过程不影响 AMG 的整体性能。

4.2.3 程序整体分析

最后，我们对 AMG-GPU 的整体性能给出详细的剖析（见附录）。在 4.1 部分的实验结果中，AMG-GPU 在给定容忍误差为 1e-6 时，性能提升效果不明显。从下表中可以看出，在向下粗化过程和插值过程中，Relax 操作（类 SpMV）和粗化、插值操作（SpMV）只在细网格层有性能提升，而当网格变粗时，甚至出现负加速比，造成了整体 AMG 性能的下降。

5 结论

本文对求解稀疏代数方程组代数多重网格(AMG)在 GPU 上进行优化，相对串行 AMG 有 2 倍的性能提升，但在 1e-6 的误差精度限制下，AMG-GPU 实现性能提升效果不明显，仅相对串行 AMG 提升约 6%。我们对影响 AMG-GPU 的因素进行了详细剖析，得出结论：由于 AMG 各层算子（操作算子、插值算子、粗化算子）规模的变化使得 SpMV/类 SpMV kernel 在 GPU 上的性能受到影响。当矩阵规模减小时，GPU 不能启动足够多的线程来掩盖访存延迟，导致性能 SpMV/类 SpMV 性能严重下降，从而导致 AMG-GPU 整体性能的下降。下一步工作我们将考虑区别对待不同规模的网格层，如只对足够细的网格层在 GPU 上进行优化，针对不同层次的特点进行自适应的调优。

致谢 在此,我们向对本文的工作给予支持和建议的老师和同学们表示感谢!

References:

[1] NVIDIA Tesla C2070: http://www.nvidia.com/object/personal-supercomputing.html

[2] William L. Briggs, Van Emden Henson Steve F. McCormick "A Multigrid Tutorial"(Second Edition) 2000

[3] R. D. Falgout “An Introduction to Algebraic Multigrid" 2006

[4] Van Emden Henson, Ulrike Meier Yang “BoomerAMG: A parallel algebraic multigrid solver and preconditioner “Applied Numerical Mathematics 41 (2002) 155–177

[5] Hypre library in LLNL https://computation.llnl.gov/casc/linear_solvers/overview.html

[6] Top 500 SuperComputer lists: http://www.top500.org/

[7] "Efficient Sparse Matrix-Vector Multiplication on CUDA”Nathan Bell etc. NVIDIA TR 2008

附录

AMG-GPU VS CPU								
时间(ms)			CPU		GPU		**加速比**	
	level	对应规模	Relax 时间	粗化过程	Relax 时间	粗化过程	**Relax 时间**	**粗化过程**
向下粗化过程	0	1000000	26.000	25.700	7.19	5.65	**3.62**	**4.55**
	1	500000	13.120	14.360	6.15	6.94	**2.13**	**2.07**
	2	250000	16.650	12.850	16.03	13.32	**1.04**	**0.96**
	3	125000	6.620	6.820	8.45	8.45	**0.78**	**0.81**
	4	62006	6.150	4.690	7.02	6.85	**0.88**	**0.68**
	5	31001	2.950	2.520	4.16	4.49	**0.71**	**0.56**
	6	7817	0.410	0.300	0.74	0.74	**0.55**	**0.41**
	7	1986	0.100	0.080	0.39	0.34	**0.26**	**0.24**
	8	565	0.040	0.020	0.39	0.3	**0.10**	**0.07**
	9	201	0.013	0.011	0.28	0.2	**0.05**	**0.06**
	10	80	0.006	0.005	0.24	0.17	**0.03**	**0.03**
	11	31	0.003	0.004	0.24	0.15	**0.01**	**0.02**
	12	12	0.002	0.002	0.22	0.15	**0.01**	**0.02**
	13	3	0.012	-	0.06		**0.20**	**-**
				插值过程		插值过程		**插值过程**
向上插值过程	13	3	0.012	0.001	0.06	0.03	**0.20**	**0.03**
	12	12	0.001	0.001	0.22	0.03	**0.01**	**0.03**
	11	31	0.002	0.001	0.23	0.03	**0.01**	**0.04**
	10	80	0.004	0.002	0.24	0.03	**0.02**	**0.07**
	9	201	0.010	0.004	0.29	0.03	**0.03**	**0.13**
	8	565	0.024	0.010	0.38	0.03	**0.06**	**0.33**
	7	1986	0.080	0.034	0.39	0.04	**0.21**	**0.85**
	6	7817	0.310	0.140	0.74	0.06	**0.42**	**2.33**
	5	31001	2.930	0.370	4.14	0.1	**0.71**	**3.70**
	4	62006	6.130	0.760	7.02	0.15	**0.87**	**5.07**
	3	125000	6.610	1.520	8.43	0.29	**0.78**	**5.24**
	2	250000	16.680	3.050	16.1	0.45	**1.04**	**6.78**
	1	500000	13.100	6.170	6.24	1.01	**2.10**	**6.11**
	0	1000000	25.850	-	7.88	-	**3.28**	**-**