定常迭代法总结报告

定常迭代法总结报告

1 算法说明

1.1 算法介绍

1.1.1 *Jacobi* 迭代法

1.1.2 *Gauss–Seidel* 迭代法

1.1.3 *SOR* 迭代法

1.2 程序输入

1.3 程序输出

1.4 算法流程

1.4.1 *Jacobi* 迭代法

1.4.2 *Gauss–Seidel* 迭代法

1.4.3 *SOR* 迭代法

2 环境描述

2.1 硬件环境

2.2 软件环境

2.3 运行环境

3 使用 CUDA 加速雅可比法

3.1 CPU 版本三种算法的总结

3.2 并行加速思路

3.3 最基础的 CUDA 加速版本

3.4 共享内存 + 循环展开

3.5 线程束洗牌 (Warp Shuffle) + 规约思想

3.6 后续进一步改进策略探究

1 算法说明

1.1 算法介绍

1.1.1 *Jacobi* 迭代法

雅可比法（*Jacobi Method*）是一种解对角元素几乎都是各行和各列的绝对值最大的值的线性方程组的算法。求解出每个对角元素并插入近似值，不断迭代直至收敛。雅克比迭代法的计算公式简单，每迭代一次只需计算一次矩阵和向量的乘法，且计算过程中原始矩阵 A 始终不变，比较容易并行计算。

1.1.2 *Gauss–Seidel* 迭代法

Gauss-Seidel 方法也称为 *Liebmann* 方法或连续位移方法，与雅可比法非常相似，同样是基于矩阵分解的原理。虽然它可以应用于对角线上具有非零元素的任何矩阵，但只能在矩阵是对角线主导的或对称的和正定的情况下，保证收敛。

1.1.3 *SOR* 迭代法

D. M. Young 于 20 世纪 70 年代提出逐次超松弛 (*Successive Over Relaxation*) 迭代法，简称 *SOR* 方法，是一种经典的迭代算法。它是为了解决大规模系统的线性等式提出来的，在 *G-S* 法基础上为提高收敛速度，采用加权平均而得到的新算法。

1.2 程序输入

针对形如 $A \cdot x = b$，我们的程序需要读入方阵 A，等号右边的向量 b，以及解向量 x 。

我们研究了很久给出的 `Equation193.stiff` 和 `Equation4800.stiff` 两个文件，但仍没有理解文件中子矩阵的分解方式，因此我们使用 Matlab 来生成实验数据。

其中，方阵 A 必须为对称正定矩阵，迭代法才能收敛。我们先生成一个随机的对角矩阵 V ，然后再随机生成一个上三角矩阵 U，然后计算 U^TVU，所得结果为对称正定矩阵 A。

然后随机生成解向量 x，使用 *Matlab* 计算向量 b，这样就能对我们计算出的结果进行验证。

将所得结果保存在 txt 文件中，作为程序的输入。除此之外，程序还要去输入矩阵大小、计算精度、输出文件名，对于超松弛迭代法，还需输入超松弛常数。

1.3 程序输出

- 迭代次数
- 计算时间
- 误差
- 结果（保存在 txt 文件里）

1.4 算法流程

1.4.1 *Jacobi* 迭代法

1. 首先将方程组中的系数矩阵 A 分解成下三角阵 L 、对角阵 D 、上三角阵型 U 三部分，即 $A = L + D + U$。
2. 然后确定迭代格式：$X^{(k+1)} = B_J X^{(k)} + f_J$

$$
\begin{cases}
x_1^{(k+1)} = \left(-a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)} + b_1\right)/a_{11} \\
x_2^{(k+1)} = \left(-a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \ldots a_{2n}x_n^{(k)} + b_2\right)/a_{22} \\
\cdots \\
x_n^{(k+1)} = \left(-a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \ldots - a_{n,n-1}x_n^{(k)} + b_n\right)/a_{nn}
\end{cases}
$$

   也即

$$
x_i^{(k+1)} = \frac{1}{a_{ii}} \sum_{j=1}^{n} a_{ij}x_j^{(k)}, i = 1, 2, \ldots, n
$$

3. 最后选取初始迭代向量 $x^{(0)}$，开始逐次迭代。

1.4.2 *Gauss–Seidel* 迭代法

与 *Jacobi* 迭代法相似，只是将 *Jacobi* 迭代法的迭代公式改为：

$$cx_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$
$$(i = 1, 2, \cdots, n; k = 0, 1, 2, \cdots, n)$$

这里假定 $a_{ii} \neq 0 (i = 1, 2, \cdots, n)$.

1.4.3 *SOR* 迭代法

设已求得 n 元线性代数方程组 $Ax = b$ 第 $k-1$次迭代向量 $x^{(k-1)} = \left(x_1^{(k-1)}, x_2^{(k-1)}, \ldots, x_n^{(k-1)} \right)^T$，及第 k 次迭代向量 $x^{(k)}$ 的分量 $x_j^{(k)} (j = 1, 2, \ldots, i-1)$，要计算分量 $x_i^{(k)}$。

1. 用 *Gauss-Seidel(GS)* 迭代求得

$$x_i' = \frac{1}{a_{ii}} \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} + b_i \right)$$

2. 计算 x_i' 与第 $k-1$ 次迭代值 $x_i^{(k-1)}$ 的加权平均 ω 作为第 k 次迭代值

$$x_i^{(k)} = (1 - \omega) x_i^{(k-1)} + \omega x_i'$$

   或者可以整理成

$$x_i^{(k)} = x_i^{(k-1)} + \frac{\omega \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i}^{n} a_{ij} x_j^{(k-1)} + b_i \right)}{a_{ii}}, i = (1, 2, \ldots, n)$$

   其中，参数 ω 称为松弛因子，$0 < \omega < 2$。当 $\omega > 1$ 时，上式称为逐次超松弛迭代法；当 $\omega = 1$ 时，上式为*Gauss-Seidel* 迭代法；当 $0 < \omega < 1$ 时，上式称为低松弛迭代法。

2 环境描述

2.1 硬件环境

- **CPU:** Intel i7-7700HQ
- **GPU:** NVIDIA GeForce GTX 1050

2.2 软件环境

- Microsoft Visual Studio Community 2017 15.9.10
- Nvidia Cuda 10.1.120

2.3 运行环境

- Visual C++ 2017

3 使用 CUDA 加速雅可比法

3.1 CPU 版本三种算法的总结

我们使用了 C++ 实现了以上三种算法的 CPU 版本，对于不同的迭代法，我们做了关于系数矩阵维度、目标精度对其收敛速度影响的实验。由于这三种算法在性能上远弱于 GPU 版本，没有太多探讨的价值，因此这里就不展示详细的结论和相关图表，只对实验结果进行总结。

高斯-赛德尔迭代、超松驰迭代基本收敛规律与雅可比迭代相同。在目标精度相同的条件下，总体来说，系数矩阵维度越大，收敛至目标精度的迭代次数越多，与此同时，系数矩阵的数据分布等自身特点也会很大程度影响到迭代次数。

相比于雅可比迭代法，高斯-赛德尔迭代和超松驰迭代收敛速度更快，迭代次数更少。但超松驰常数 ω 对于收敛速度影响很大，当 ω 取值不合适时，超松驰迭代甚至可能会远慢于前两种迭代，但 ω 的值不好确定，其最佳值求解有待研究。

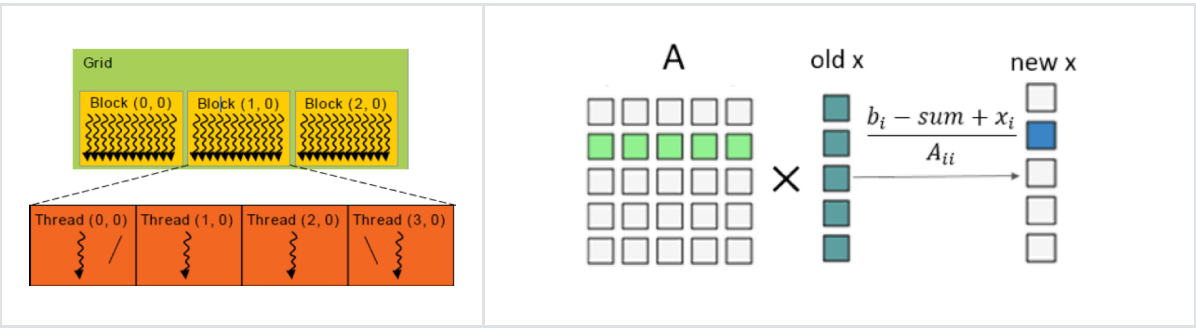
3.2 并行加速思路

以上三种算法中，由于 *Gauss-Seidel* 迭代法和 *SOR* 迭代法中每一次迭代中都需要用到上一次迭代的结果，因此可挖掘的并行性不多（实际上它们对于雅可比法的改进思路中已经包含了对其并行性的挖掘），我们着重进行了对 *Jacobi* 法的迭代实现。

雅可比迭代法中可并行计算的部分包括每轮迭代中 x 的计算以及每轮中误差的计算，其中占比最大的部分是每轮迭代中 x 的计算，因为其计算的复杂性远大于后者，所以我们选择这个部分使用并行加速进行计算。

3.3 最基础的 CUDA 加速版本

- **算法设计**：将网格设置为一维网格，其中包含若干线程块，每个线程块也设置为一维形式，其中包含若干线程，这些线程中的每个线程负责计算每轮迭代中的 x 中的一个元素，即以步长为 1 的循环遍历方式去计算 A 中的一行乘上一个 x（一列）。



- **实验结果：**

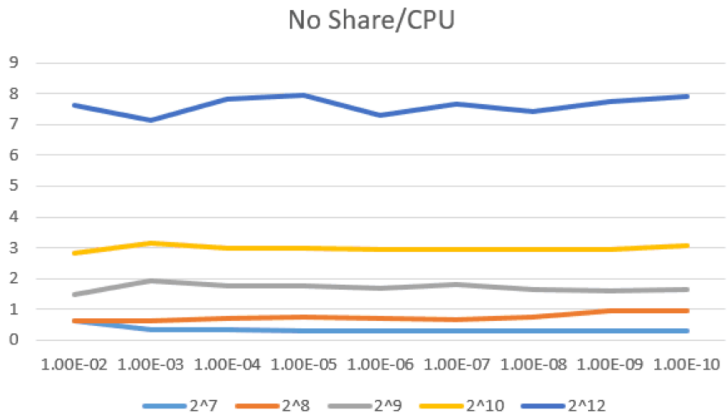


Fig. 3.3-1: CUDA 基本版本与 CPU 版本的加速比

从以上结果中可以看出，相对于 CPU 版本，该算法对于大小为 4096 的矩阵能够加速到 8 倍左右。

- **算法分析**：
  - **优点**：从 CPU 迭代算法中 x 中元素串行计算，变成为这些元素的并行计算，从而通过基本的并行加速减少计算时间；
  - **缺点**：该算法中每个线程块中的线程会多次访问位于 device 全局内存的数据，即上一轮迭代的 x，A 的一整行与 x 一整列的计算仅由一个线程负责，单个线程的负担过重，并行性不够高，效率太低。

3.4 共享内存 + 循环展开

- **算法设计**：针对上面方法存在的问题，我们将一个线程中每次步长为 1 的循环遍历计算，改为步长为线程块大小去遍历计算，在每轮循环中使用循环展开；同时为每个线程块设置一个大小与线程块大小等同的共享内存数组，然后每轮循环遍历中一个线程块中的每个线程都会并行地加载一个共享内存数组中的一个元素，从而每轮循环只需访问全局内存中的 x 一次，该轮循环中剩下需要访问 x 的时候只需从共享内存中读取。

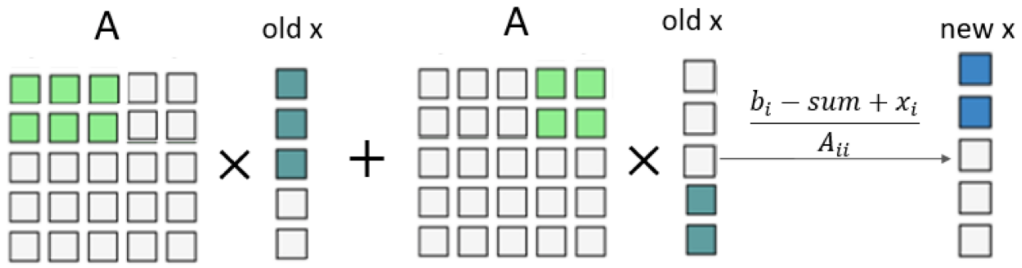


Fig. 3.4-1: 同一线程块内的线程使用共享内存的示意图（墨绿色部分为使用共享内存的部分）

- **实验结果**：

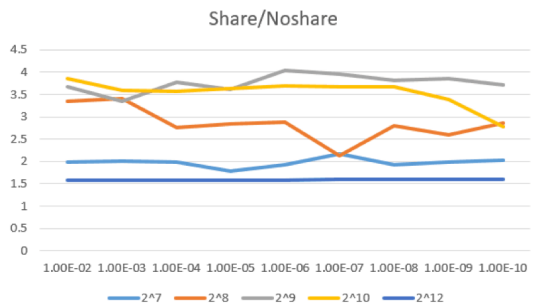


Fig. 3.4-2: 共享内存版本与基本版本的加速比

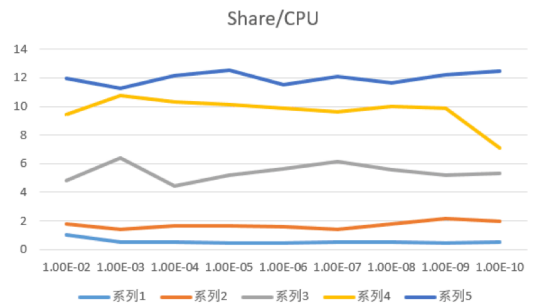


Fig. 3.4-3: 共享内存版本与 CPU 版本的加速比

从图中可以看出，当维度达到 4096 时，加速比可达 12 倍。

- **算法分析**：
  - **优点**：循环展开中的语句是没有数据相关的，从而增加了并发执行的机会；将从全局内存的读取大部分替换为从读取速度更快的块内共享内存，从而减少内存数据的读取消耗。
  - **缺点**：下一个 x 元素的计算还是仅由一个线程来完成，所以其并行性还是不够高。

3.5 线程束洗牌 (Warp Shuffle) + 规约思想

- **算法设计**：针对上面的问题，将网格还是设置为一维的，但其长度设置为与系数矩阵的行数等同，每个线程块的大小设置为 32（与 GPU 中流处理器 SM 执行基本单位线程束的大小一致），一个线程块中的线程负责 A 中的一行乘上一个 x（一列），每个线程对A中一行里间隔 32 个进行一次乘积，并累加，每个线程都并行的做乘积累加到各自的终点，然后对这 32 个线程之间通过 *Warp Shuffle* 来进行规约求和。

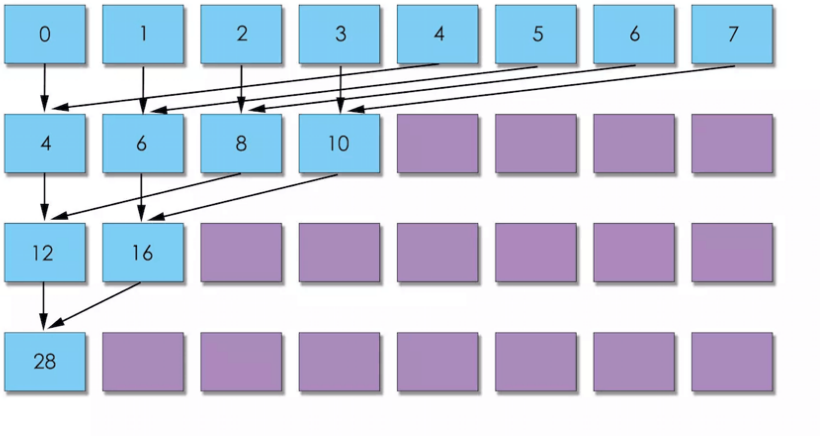


Fig. 3.5-1: 规约思想示意图

- **实验结果**：

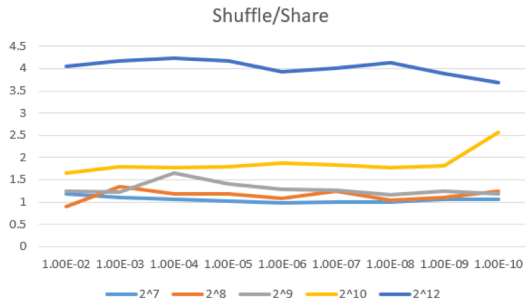


Fig. 3.5-2: 线程束洗牌与共享内存版本的加速比

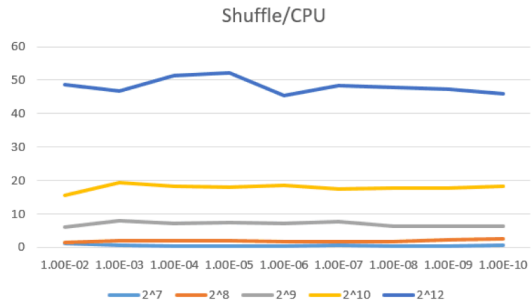


Fig. 3.5-3: 线程束洗牌版本与 CPU 版本的加速比

- **算法分析**：
  - **优点**：一个 x 元素的计算是由 32 个线程共同负责，同时 A 的一行和 x（上一个）的数据加载也是由这 32 个线程共同负责，增加了并行性；同时使用规约的方式来对 32 个线程的累加和进行进一步求和，减少了加法的次数；使用了 *Warp Shuffle* 来让线程直接读取同一线程束的其他线程的寄存器值，只有很低的延迟，同时也不消耗额外的内存资源来执行数据交换。

3.6 后续进一步改进策略探究

- 当前维度下的改进策略
  - 判断最大误差的步骤并行化
  - 使用动态并行策略，将是否迭代的条件判断放到核函数里，让核函数调用核函数，从而使得整个算法全在核函数内执行（难度很大）
- 系数矩阵A的维度达到几万甚至十几万更高时
  - GPU的显存已经无法放下这些数据，需要采用分批计算的策略，这个时候就可以使用多GPU或者分布式GPU进行并行加速计算。