计算机科学
COMPUTER SCIENCE

基于 GPU 的并行 DILU 预处理技术

汪晋，刘江

引用本文

汪晋，刘江. 基于 GPU 的并行 DILU 预处理技术[J]. 计算机科学, 2022, 49(6): 108-118.

WANG Jin, LIU Jiang. GPU-based Parallel DILU Preconditioning Technique[J]. Computer Science, 2022, 49(6): 108-118.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

OpenFoam 中多面体网格生成的 MPI + OpenMP 混合并行方法

Hybrid MPI+OpenMP Parallel Method on Polyhedral Grid Generation in OpenFoam

计算机科学, 2022, 49(3): 3-10. https://doi.org/10.11896/jsjkx.210700060

基于逐次超松弛技术的 Double Speedy Q-Learning 算法

Double Speedy Q-Learning Based on Successive Over Relaxation

计算机科学, 2022, 49(3): 239-245. https://doi.org/10.11896/jsjkx.201200173

基于代价敏感卷积神经网络的非平衡问题混合方法

Cost-sensitive Convolutional Neural Network Based Hybrid Method for Imbalanced Data Classification

计算机科学, 2021, 48(9): 77-85. https://doi.org/10.11896/jsjkx.200900013

基于 GPU 的特征脸算法优化研究

Optimization of GPU-based Eigenface Algorithm

计算机科学, 2021, 48(4): 197-204. https://doi.org/10.11896/jsjkx.200600033

基于 GPU 加速的并行 WMD 算法

Parallel WMD Algorithm Based on GPU Acceleration

计算机科学, 2021, 48(12): 24-28. https://doi.org/10.11896/jsjkx.210600213

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210300259

基于 GPU 的并行 DILU 预处理技术

汪 晋 刘 江
1 中国科学院重庆绿色智能技术研究院 重庆 400714
2 中国科学院大学计算机科学与技术学院 北京 100049
(wangjin@cigit.ac.cn)

摘 要 在科学计算和工程领域,大型稀疏线性方程组的求解非常常见,目前已经有许多迭代方法和预处理技术被用于求解这类方程。DILU 预处理技术类似于 ILU,是开源计算流体力学软件 OpenFOAM 中重要的预处理技术,但未在 OpenFOAM 以外的领域引起关注,目前也没有完整的 GPU 实现。比较了 DILU 和 ILU 预处理技术对稳定双共轭梯度法(BiCGStab)加速的效果,以及它们在构造预处理子上的开销,结果表明,DILU 在加速效果上不逊于 ILU 且在稳定性上优于 ILU。在 GPU 并行实现方面,DILU 可以使用分层并行和无全局同步并行两种并行策略,详细讨论了 DILU 预处理技术在这两种策略下的实现方法,给出了相关的算法和参考代码,然后比较了在两种并行策略下 DILU 预处理技术的性能。数值实验结果表明,在实践中两种并行策略各有优劣,可以根据实际表现进行选择。另外比较了 GPU 和 CPU 执行的 DILU 预处理技术,GPU 在性能上具有明显优势,在线性方程组求解上存在性能瓶颈的程序可以移植到 GPU 平台以提升性能。

关键词:GPU;CUDA;预处理;DILU;稀疏三角方程;迭代法;OpenFOAM
中图法分类号 TP391.9

GPU-based Parallel DILU Preconditioning Technique

WANG Jin and LIU Jiang
1 Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing 400714, China
2 School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract Large sparse linear equations often appear in scientific computation and engineering. There are many iterative methods and preconditioning techniques for solving these linear equations. Diagonal-based incomplete LU (DILU) is a preconditioning technique similar to incomplete LU (ILU) factorization. DILU is applied in OpenFOAM, an open source computational fluid dynamics software, and is a very important preconditioning technique in OpenFOAM. DILU has not received extensive attention outside OpenFOAM, and there is no complete GPU-based implementation so far. This paper compares DILU preconditioned BiCGStab with ILU preconditioned BiCGStab, and the time elapses in preconditioner constructions. The numeric experiments suggest that DILU may be more efficient and stable than ILU. As for GPU-based parallel implementations, this paper discusses two parallel schemes, that are level-set scheme and synchronization-free scheme, and gives related algorithms and some codes under these two parallel schemes. It compares the performances of DILU preconditioning technique under two parallel schemes. The numeric results show that each scheme has its own advantages and disadvantages in different equations, and we can select one according to their performances in practice. This paper compares the performance of DILU preconditioning on GPU and CPU, and the results show that GPU is more competitive. The applications that have performance bottlenecks on linear systems solutions can be improved by moving to GPU platforms.

Keywords GPU, CUDA, Preconditioning, DILU, Sparse triangular equations, Iterative methods, OpenFOAM

1 概述

1.1 稀疏线性方程组概述

在科学研究和工程领域,一般通过数值方法对微分方程进行求解。常见的微分方程数值求解方法包括有限差分法、有限单元法和有限体积法[1],这些方法的思路都是在计算区域将微分方程进行离散,得到一个线性方程组。从微分方程离散得到的线性方程组具有稀疏的特点,即相比方程的规模,

到稿日期:2021-03-25 返修日期:2021-07-27
基金项目:国家自然科学基金(61672488);国家重点研究发展计划(2018YFC0116704)
This work was supported by the National Natural Science Foundation of China (61672488) and National Key R&D Program of China (2018YFC0116704).
通信作者:刘江(liujiang@cigit.ac.cn)

其非零元素非常少；并且在许多问题中，为达到理想要计算精度，计算区域划分的网格必须十分精细，得到的线性方程组的规模也会非常大。综上，在使用线性方程组求解算法时，我们常常需要面对大规模稀疏线性方程组。此外，许多涉及特征值的数值问题的求解思路与线性方程组的求解思路相同[2]，因此如何高效求解大规模稀疏线性方程组极具研究价值。

由于稀疏线性方程组具有规模大和稀疏的特点，在实践中一般使用专门设计的存储方式来对其进行存储，常见的存储方法有 CSR(Compressed Sparse Row)，COO(Coordinate)和 DIA(Diagonal)等[3]。相比传统的稠密存储方法，稀疏存储方法可以有效地节省存储空间，但无法灵活地访问矩阵元素，矩阵操作如矩阵乘向量无法使用 BLAS(Basic linear Algebra Subroutine)定义的接口，必须根据存储方式重新设计算法。

稀疏线性方程组的求解方法分为两类：直接法[4]和迭代法[1,5]。直接法通过矩阵分解(如 LU 分解和 Cholesky 分解)将方程分解为两个三角方程，然后求解三角方程得到方程的解。直接法的优点是求解结果的精度高、性能稳定；缺点是对存储空间的需求较大。在稀疏存储方式下，直接法计算过程中会将一些零元素变成非零元素，导致矩阵分解的结果往往比较稠密，在面对规模非常大的稀疏线性方程组时常常无能为力。迭代法从一个初始解向量出发，经过一系列运算得到下一个近似解向量，然后反复进行这个过程，近似解向量可以逐渐收敛到精确解。迭代法的优点是不更改系数矩阵本身，因此其对存储空间的需求较低，几乎是求解大规模稀疏线性方程组的唯一方法。

常见的迭代法有 Jacobi 迭代、共轭梯度法(Conjugate Gradient，CG)和双共轭梯度法(BiConjugate Gradient，BiCG)等[1,5]。迭代法的核心操作是矩阵乘向量，即 SpMV，高效地实现这个操作是提高迭代法性能的关键之一，目前已有许多文献讨论了它在 GPU 上的实现[6-9]。

迭代法的收敛速度一般与方程的系数矩阵的特征值分布情况有关，特征值病态分布的方程使用迭代法求解时收敛速度非常慢。为改善系数矩阵的特征值分布情况，提出了许多预处理技术。经典的预处理技术有不完全 LU 分解(Incomplete LU，ILU)、不完全 Cholesky 分解(Incomplete Cholesky，IC)和稀疏近似逆(Sparse Approximate Inverse)等[1]。开源计算流体力学软件 OpenFOAM[10-12]中有两种非常重要的预处理技术：DILU (Diagonal-based Incomplete LU) 和 DIC (Diagonal-based Incomplete Cholesky)。这两种预处理技术具有构造简单和数值稳定性良好的优点，分别适用于一般方程组和对称方程组，但没有在 OpenFOAM 以外的领域引起广泛关注。本文比较了 DILU 和 ILU 对迭代法的加速效果以及它们的性能，结果表明，DILU 在数值稳定性上更具优势，可以应用到更广泛的领域。

1.2　GPU 并行计算概述

传统的并行计算机由多处理器系统组成，通用处理器(Central Processing Units，CPU)缓存容量大，具有乱序执行、多发射流水线和分支预测等功能，在执行逻辑复杂的代码时具有较高的吞吐量。然而，在科学计算和工程领域，许多问题的计算过程中，数学运算的耗时远高于逻辑控制，使用通用处理器科简值是大材小用，因此许多协处理器应运而生，专门用于处理数学运算，GPU(Graphics Processing Units)是其中非常典型的一种。

与 CPU 相比，GPU 运算单元功能简单，无乱序执行、分支预测等功能，但其运算单元数量巨大，在此优势下，一些并行任务可以获得很高的计算速度。早期的 GPU 以图形渲染能力为主，在 2007 年，NVIDIA 推出 CUDA(Compute Unified Device Architecture)，使得开发人员可以使用 C 语言编写运行于 GPU 的程序，直接控制 GPU 的硬件资源，并且 GPU 可以用于通用计算(General-Purpose Computing on Graphics Processing Units，GPGPU)[13]。

GPU 具有浮点运算峰值高、内存带宽高、存储区域类型多、寄存器数量大等优点，非常适合用于稀疏线性方程组求解，目前已有许多文献讨论了迭代方法和预处理方法在 GPU 上的实现。本文关注 OpenFOAM 中的 DILU 预处理方法在 GPU 上的实现，并比较了 DILU 和 ILU 在性能上的差异，希望能为预处理技术的选择和 DILU 预处理技术的 GPU 并行实现提供一些参考。

1.3　相关工作

1977 年，Meijerink 等[14]提出了使用不完全 Cholesky 分解的预处理共轭梯度法(Incomplete Cholesky Conjugate Gradient，ICCG)，这是第一种基于矩阵分解的预处理技术，明确了可以将矩阵的不完全分解用作迭代法的预处理子。1978 年，Kershaw[15]分析了不完全 Cholesky 分解预处理技术对共轭梯度法的作用，推广了不完全分解在预处理技术方面的应用。目前，流行的基于不完全分解的预处理技术还有 ILU、具有填充层级的不完全 LU 分解(ILU(l))和带舍弃阈值的不完全 LU 分解(Incomplete LU Factorization with Threshold，ILUT)等[16]。此外，开源流体力学计算软件 OpenFOAM 中也有重要的预处理技术 DILU 和 DIC[10-11,17]。

在预处理技术的 GPU 实现方面，有大量文献讨论了 ILU 预处理技术在 GPU 上的实现和稀疏三角方程在 GPU 上的并行求解。Anderson 等[18]提出利用层次信息并行求解稀疏三角方程的经典方案。Naumov[19]于 2011 年提出了利用层次信息的 GPU 并行的稀疏三角方程组求解算法，解决了并行预处理技术中的一大难题。2013 年，Li 等[20]在 GPU 平台上实现了预处理共轭梯度法，并实现了 ILU(0)预处理技术。2015 年，Naumov 等[21]讨论了矩阵着色算法的 GPU 并行实现，并研究了不完全 LU 分解的 GPU 并行实现。以上这些实现都是基于分层并行策略。2016 年，Liu 等[22]指出分层并行的方法在层次间的全局同步上开销较大，并提出了一种无全局同步的稀疏三角方程并行求解算法，这种算法在 GPU 上工作时必须用 32 个线程求解一个分量。2017 年，Li[23]总结了 GPU 并行求解稀疏三角方程的技术。2020 年，Su 等[24]提出了一种新的无全局同步的稀疏三角方程并行求解算法，且在 GPU 上工作时可以用一个线程求解一个分量。此外，更多这方面的研究可参考文献[25-28]。

然而，作为 OpenFOAM 中重要的预处理技术，DILU 和 DIC 并未在 OpenFOAM 以外的领域引起广泛关注，且 Open-

FOAM本身使用MPI进行并行计算，进程内部由CPU串行执行，无法利用GPU进行加速。目前有一些工作尝试为OpenFOAM引入GPU并行的线性方程组求解器[29-30]，但没有见到DILU和DIC预处理技术的完整实现，为此本文提供了比较完整的解决方案。由于DIC预处理技术实际上就是DILU的对称版本，因此本文只关注如何实现DILU预处理技术，DIC的实现与此类似。

2 并行DILU算法

2.1 DILU预处理技术

对于稀疏线性方程组 $\boldsymbol{Ax}=\boldsymbol{b}$，如果构造预处理子 $\boldsymbol{P}=\boldsymbol{P}_1\boldsymbol{P}_2$，令 $\bar{\boldsymbol{A}}=\boldsymbol{P}_1^{-1}\boldsymbol{A}\boldsymbol{P}_2^{-1}$，$\bar{\boldsymbol{x}}=\boldsymbol{P}_2\boldsymbol{x}$，$\bar{\boldsymbol{b}}=\boldsymbol{P}_1^{-1}\boldsymbol{b}$，可得预处理方程：

$$\bar{\boldsymbol{A}}\bar{\boldsymbol{x}}=\bar{\boldsymbol{b}} \tag{1}$$

若 $\boldsymbol{P}_1$ 取单位矩阵 $\mathbf{I}$，则为右预处理，若 $\boldsymbol{P}_2$ 取单位矩阵 $\mathbf{I}$，则为左预处理，如果 $\mathbf{P}_1$ 和 $\mathbf{P}_2$ 都不为 $\mathbf{I}$ 则为两边预处理。

不同的迭代法在进行预处理时可能采取不同的预处理方式，如共轭梯度法的3种预处理方式得到的算法是一样的，而广义极小残量法[31]（Generalized Minimal Residual，GMRES）和稳定双共轭梯度法[32]（Biconjugate Gradient Stabilized，BiCGStab）分别采用右预处理和两边预处理。

DILU预处理技术适用于一般方程，不要求方程对称，亦不要求方程正定，因此一般配合GMRES或者BiCGStab使用。DIC为DILU的对称版本，利用方程的对称性做了一定的优化，可以配合适用于对称正定方程组的共轭梯度法使用。预处理稳定双共轭梯度法如算法1所示。

算法1 预处理稳定双共轭梯度法[32]

输入：n×n系数矩阵 **A**；n×1右侧向量 **b**；n×n预处理子 **P**

输出：n×1解向量 **x**

1. $\mathbf{x}_0$ is an initial guess. $\mathbf{r}_0=\mathbf{b}-\mathbf{Ax}$.
2. $\bar{\mathbf{r}}_0$ is an arbitrary vector, such that $(\mathbf{r}_0,\bar{\mathbf{r}}_0)\neq 0$, e.g. $\bar{\mathbf{r}}_0=\mathbf{r}_0$.
3. $\rho_0=\alpha_0=\omega_0=1$. $\mathbf{v}_0=\mathbf{p}_0=\mathbf{0}$.
4. for i=1,2,3,… do
5. if $\mathbf{x}_{i-1}$ is accurate enough, then break
6. $\rho_i=(\bar{\mathbf{r}}_0,\mathbf{r}_{i-1})$. $\beta=(\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$.
7. $\mathbf{p}_i=\mathbf{r}_{i-1}+\beta(\mathbf{p}_{i-1}-\omega_{i-1}\mathbf{v}_{i-1})$.
8. $\mathbf{y}=\mathbf{P}^{-1}\mathbf{p}_i$. $\mathbf{v}_i=\mathbf{Ay}$. // 预处理
9. $\alpha_i=\rho_i/(\bar{\mathbf{r}}_0,\mathbf{v}_i)$. $\mathbf{s}=\mathbf{r}_{i-1}-\alpha_i\mathbf{v}_i$.
10. $\mathbf{z}=\mathbf{P}^{-1}$ s. t $=\mathbf{Az}$. // 预处理
11. $\omega_i=(\mathbf{t},\mathbf{s})/(\mathbf{t},\mathbf{t})$.
12. $\mathbf{x}_i=\mathbf{x}_{i-1}+\alpha_i\mathbf{y}+\omega_i\mathbf{z}$. $\mathbf{r}_i=\mathbf{s}-\omega_i\mathbf{t}$.
13. end for

不同的预处理技术的主要区别在于如何选择预处理子 $\boldsymbol{P}$。ILU预处理技术对方程系数矩阵 $\boldsymbol{A}$ 做LU分解，且在消元过程中跳过所有零元素。DILU预处理技术将系数矩阵 $\boldsymbol{A}$ 分裂为严格下三角部分 $\boldsymbol{L}$、严格上三角部分 $\boldsymbol{U}$ 和对角部分 $\boldsymbol{D}$ 3部分，且 $\boldsymbol{A}=\boldsymbol{L}+\boldsymbol{D}+\boldsymbol{U}$；然后构造预处理子（Preconditioner）。

$$\boldsymbol{M}=(\boldsymbol{E}+\boldsymbol{L})\boldsymbol{E}^{-1}(\boldsymbol{E}+\boldsymbol{U}) \tag{2}$$

其中，$\boldsymbol{E}$ 是一个对角矩阵，满足：

$$\mathrm{diag}(\boldsymbol{A})=\mathrm{diag}(\boldsymbol{M}) \tag{3}$$

由于：

$$\mathrm{diag}(\boldsymbol{M})=\mathrm{diag}(\boldsymbol{E}+\boldsymbol{L}\boldsymbol{E}^{-1}\boldsymbol{U}+\boldsymbol{L}+\boldsymbol{U})=\mathrm{diag}(\boldsymbol{E}+\boldsymbol{L}\boldsymbol{E}^{-1}\boldsymbol{U}) \tag{4}$$

容易得到 $\boldsymbol{E}$ 的计算方法为：

$$\begin{cases}\boldsymbol{E}_{11}=\boldsymbol{A}_{11}\\ \boldsymbol{E}_{22}=\boldsymbol{A}_{22}-\boldsymbol{L}_{21}\boldsymbol{E}_{11}^{-1}\boldsymbol{U}_{12}\\ \quad\vdots\\ \boldsymbol{E}_{ii}=\boldsymbol{A}_{ii}-\sum_{j=1}^{i-1}\boldsymbol{L}_{ij}\boldsymbol{E}_{jj}^{-1}\boldsymbol{U}_{ji}\end{cases} \tag{5}$$

容易看出DILU预处理子的计算过程是完全串行的，在GPU这样高度并行的硬件上实现它难度非常大。本文给出了利用层次信息并行和无全局同步并行两种并行策略下，DILU预处理子的并行构造方法。

在实施预处理时，即在算法1的第8行和第10行，主要为求 $\boldsymbol{P}^{-1}$ 与一个向量的乘积，直接求 $\boldsymbol{P}^{-1}$ 是不可行的，因为时间复杂度太高，并且稀疏矩阵的逆一般不是稀疏矩阵，在存储空间的需求上太高。在实践中，一般通过求解两个三角方程来得到 $\boldsymbol{P}^{-1}$ 与向量之积。

将DILU预处理用于预处理稳定双共轭梯度法，需要计算 $\boldsymbol{M}^{-1}$ 与向量之积，采用解三角方程的方法来计算。假设需要计算 $\boldsymbol{M}^{-1}\boldsymbol{f}$，即求向量 $\boldsymbol{g}$，满足 $\boldsymbol{Mg}=\boldsymbol{f}$，即求解式(6)：

$$(\boldsymbol{E}+\boldsymbol{L})\boldsymbol{E}^{-1}(\boldsymbol{E}+\boldsymbol{U})\boldsymbol{g}=\boldsymbol{f} \tag{6}$$

先解下三角方程：

$$(\boldsymbol{E}+\boldsymbol{L})\boldsymbol{g}_1=\boldsymbol{f} \tag{7}$$

再解上三角方程：

$$(\boldsymbol{E}+\boldsymbol{U})\boldsymbol{g}=\boldsymbol{E}\boldsymbol{g}_1 \tag{8}$$

即可得到待求向量 $\boldsymbol{M}^{-1}\boldsymbol{f}$。

三角方程的求解可以使用前代法和回代法，但它们是完全串行的算法，与DILU预处理子的构造类似。本文给出了利用层次信息并行和无全局同步两种并行策略下的稀疏三角方程并行求解方法，并比较了两种策略下预处理的性能。

2.2 利用层次信息并行的算法

2.2.1 三角矩阵着色

在利用层次信息并行的算法中，首先需要计算层次信息，这一步一般被称为矩阵着色，用于发掘并行性。矩阵着色[33]实际上是对矩阵对应的图进行着色，这是图论中一个比较经典的问题。文献[18,34]利用矩阵着色发掘三角方程求解过程中的并行性，提出了经典的分层并行的三角方程求解方法。文献[21]讨论了矩阵着色算法的GPU实现，并将它用于ILU预处理技术的GPU并行实现。下面介绍并行的矩阵着色算法。

考虑一个 $n\times n$ 的稀疏三角矩阵 $\boldsymbol{T}$，它对应一个有向无环图 $G(V,E)$，$V=\{v_0,v_1,\cdots,v_{n-1}\}$ 是顶点集，边集 E 由以下规则定义：如果 $\boldsymbol{T}_{ij}\neq 0$，则图中存在一条由顶点 v_j 指向顶点 v_i 的边 (v_j,v_i)。以下三角矩阵 $\boldsymbol{L}_1$ 为例：

$$\boldsymbol{L}_1=\begin{bmatrix}0&&&&&&\\ &1&&&&&\\ &1&2&&&&\\ &1&&3&&&\\ 1&&1&&4&&\\ &&1&1&&5&\\ &1&&&&&6\end{bmatrix} \tag{9}$$

它对应的有向无环图如图 1 所示。上三角矩阵与此类似。

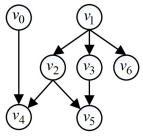


图 1　$\boldsymbol{L}_1$ 对应的有向无环图

Fig. 1　Directed acyclic graph of $\boldsymbol{L}_1$

矩阵着色就是标记每个顶点所在的层数，且如果存在一条从 v_i 指向 v_j 的边，则 v_j 对应的层数必须大于 v_i 对应的层数。其基本思路是从图中选出入度为 0 的所有顶点加入最大独立集，然后标记这些节点的层次为轮次，再将它们从图中移除，重复这个过程直到图为空。以 $\boldsymbol{L}_1$ 为例，其着色的结果为 $levels=\{0,0,1,1,2,2,1\}$，如图 2 所示。

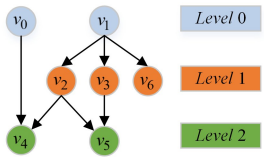


图 2　$\boldsymbol{L}_1$ 的着色结果

Fig. 2　Coloring result of $\boldsymbol{L}_1$

这样的结果有一个缺点，即输出的结果数组 *levels* 中，仅有各个顶点的层次信息，同一层的顶点不是连续分布的，使用时如果要挑出某一层的所有顶点，就需要遍历数组 *levels*。为解决这个问题，可以以 *levels* 为键，对顶点进行排序，然后对 *levels* 进行压缩，得到各个层次在排序后的顶点数组中的起始偏移。

对 $\boldsymbol{L}_1$ 的着色结果进行重排序和压缩，结果如图 3 所示。第 0 层在 *ordered* 中的下标范围为[0,2)，第 1 层在 *ordered* 中的下标范围为[2,5)，第 2 层在 *ordered* 中的下标范围为[5,7)，数组 *levoff* 的最后一个元素是哨兵，保证从偏移数组 *levoff* 中取出上述范围时不越界。

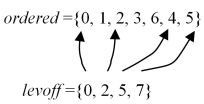


图 3　$\boldsymbol{L}_1$ 着色后再排序的结果

Fig. 3　Reordering vertices of $\boldsymbol{L}_1$ by levels

有向无环图着色的过程可以参考文献[21]。将有向无环图着色算法与稀疏矩阵相结合，并添加顶点重排序的过程，可以得到实用的三角矩阵并行着色算法，如算法 2 所示。

算法 2　三角矩阵着色算法(ColorTriangular)

输入：n×n 稀疏三角矩阵 **T**

输出：整数 levcnt，表示层次总数；整数数组 ordered，表示排序后的顶点；整数数组 levoff，表示各层在 ordered 中的起始偏移

```
1. Initializelevels={+∞,+∞,…} // 所有顶点未着色
2. for lev=0 to n-1 do
3. if any elements of levels is not+∞,then break // 着色完毕
4. parallel for i=0 to n-1 do
5. if levels[i]≠+∞,then continue // 已经着色
6. for j:j≠i and T(i,j)≠0 do
7.     if levels[j] ≥ lev,// 前驱未着色或本轮着色
8.         then go to (4) to start the next loop
9.     end for
10. levels[i]=lev
11.   end for
12. end for
13. levcnt=lev
14. Initialize ordered={0,1,2,…,n-1}
15. ParallelSortByKey(key=levels,value=ordered)
16. levoff[0]=0,levoff[levcnt]=n
17. parallel for i=0 to n-1 do
18.     if levels[i]==levels[i-1],then continue
19.     levoff[levels[i]]=i
20. end for
```

算法 2 在访问数组 *levels* 时存在数据竞争(第 7 行读，第 10 行写，读取的结果仅在第 8 行使用)，考虑一个线程在读 *levels*[*j*]时，另一个线程在写 *levels*[*j*]，那么读线程将会读到什么样的结果取决于读写实际发生的顺序。算法 2 并未使用互斥机制，那么读写顺序是无法预测的。但在算法 2 中，写线程写成功之前 *levels*[*j*]必然为+∞，写成功之后为 *lev*，读线程只会读取到+∞和 *lev* 中的一个，而无论读到哪一个都不会影响第 7 行的判断结果，因此算法 2 不需要使用互斥机制即可保证计算结果的正确。

算法 2 的第 15 行涉及并行排序算法，已经有许多论文提出了很多行之有效的方法，如奇偶排序(Batcher's Odd-even Mergesort)、双调排序(Bitonic Mergesort)等[35-36]。这里可以直接使用 CUDA 的模板库函数 thrust::sort_by_key 在 GPU 上进行并行排序。

2.2.2　构造

在计算对角矩阵 $\boldsymbol{E}$ 的第 i 个元素 $\boldsymbol{E}_{ii}$ 时，依赖于 $\boldsymbol{E}_{11}$ 到 $\boldsymbol{E}_{i-1,i-1}$，因此 $\boldsymbol{E}$ 的计算过程是完全串行的。如果系数矩阵是一个稀疏矩阵，则 $\boldsymbol{E}_{ii}$ 的计算仅依赖于 $\boldsymbol{E}_{11}$ 到 $\boldsymbol{E}_{i-1,i-1}$ 中 $\boldsymbol{L}_{ij}$ 和 $\boldsymbol{U}_{ji}$ 都不为 0 的 $\boldsymbol{E}_{jj}$，因此某些 $\boldsymbol{E}_{ii}$ 是可以同时计算的，哪些可以同时计算也可以使用矩阵着色算法来发掘。本文使用 Hadamard 积[37]简化 $\boldsymbol{E}$ 的计算过程。令 $\boldsymbol{K}$ 为 $\boldsymbol{L}$ 和 $\boldsymbol{U}^{\mathrm{T}}$ 的 Hadamard 积，即 $\boldsymbol{L}$ 和 $\boldsymbol{U}^{\mathrm{T}}$ 的对应元素相乘，然后利用 $\boldsymbol{K}$ 计算 $\boldsymbol{E}$ 的过程为：

$$\begin{cases}\boldsymbol{E}_{11}=\boldsymbol{A}_{11}\\ \boldsymbol{E}_{22}=\boldsymbol{A}_{22}-\boldsymbol{K}_{12}\boldsymbol{E}_{11}^{-1}\\ \quad\vdots\\ \boldsymbol{E}_{ii}=\boldsymbol{A}_{ii}-\sum_{j=1}^{i-1}\boldsymbol{K}_{ij}\boldsymbol{E}_{jj}^{-1}\end{cases}$$

对 $\boldsymbol{K}$ 进行着色得到层次信息后，可以按层计算对角矩阵 $\boldsymbol{E}$，同层顶点对应的元素可以同时计算，且它们仅依赖层次更小的元素，因此可以按照层次从小到大进行计算，计算过程如算法 3 所示。

算法 3　DILU 分层并行构造算法

输入：n×n 严格下三角矩阵 **L**；n×n 严格上三角矩阵 **U**；n×n 对角矩阵 **D**

输出：n×n 对角矩阵 **E**

```
1. K=Hadamard(L,U^T)
2. {levcnt,ordered,levoff}=ColorTriangular(K)
3. for lev=0 to levcnt-1 do
```

4. parallel for off=levoff[lev] to levoff[lev+1]−1 do
5. i=ordered[off]
6. $\mathbf{E}(i,i)=\mathbf{D}(i,i)-\sum_{j=1}^{i-1}\mathbf{K}_{ij}\mathbf{E}_{jj}^{-1}$
7. end for
8. end for

计算出预处理子 $\boldsymbol{M}$ 分解形式中的 $\boldsymbol{E}$,$\boldsymbol{L}$ 和 $\boldsymbol{U}$ 以后,不需要再通过矩阵乘法得到 $\boldsymbol{M}$,用它的分解形式即可进行预处理。

2.2.3 预处理

预处理实施时主要求解两个稀疏三角方程。与算法 3 类似,求解稀疏三角方程可以通过按层计算来进行并行,求解过程如算法 4 所示。在算法 4 的第 4 行中,如果 $\boldsymbol{T}$ 是下三角矩阵,则求和范围为[1,i−1],如果 $\boldsymbol{T}$ 是上三角矩阵,则求和范围为[i+1,n]。如果矩阵采用稀疏存储方式,则只需要遍历这一行的非零元素,可大大降低时间复杂度。

算法 4 稀疏三角方程分层并行求解算法[18]

输入:n×n 稀疏三角矩阵 **T**;n×1 右侧向量 **b**;整数数组 ordered;整数数组 levoff;总层数 levcnt

输出:n×1 结果向量 **x**

1. for lev=0 to levcnt−1 do
2. parallel for off=levoff[lev] to levoff[lev+1]−1 do
3. i=ordered[off]
4. $\mathbf{x}_i=(\mathbf{b}_i-\sum_{\substack{j=1\\j\neq i}}^{n}\mathbf{T}_{ij}\mathbf{x}_j)/\mathbf{T}_{ii}$
5. end for
6. end for

2.2.4 小结

总的来讲,DILU 预处理技术分为两个阶段:构造阶段和预处理阶段。分层并行策略下,构造阶段按照以下流程进行:

(1)取出系数矩阵 $\boldsymbol{A}$ 的严格下三角部分作为 $\boldsymbol{L}$,严格上三角部分作为 $\boldsymbol{U}$,对角部分作为 $\boldsymbol{D}$;

(2)调用算法 3 计算预处理子 $\boldsymbol{M}$ 中的对角矩阵 $\boldsymbol{E}$;

(3)计算 $\bar{\boldsymbol{L}}=\boldsymbol{E}+\boldsymbol{L}$ 和 $\bar{\boldsymbol{U}}=\boldsymbol{E}+\boldsymbol{U}$;

(4)调用算法 2 分别对 $\bar{\boldsymbol{L}}$ 和 $\bar{\boldsymbol{U}}$ 进行着色。

预处理阶段在迭代的过程中进行,主要为计算 $\boldsymbol{M}^{-1}\boldsymbol{f}$,这一步可能需要对不同的向量 $\boldsymbol{f}$ 进行多次计算,因此应当将涉及的稀疏三角矩阵的着色放在构造阶段,计算 $\boldsymbol{M}^{-1}\boldsymbol{f}$ 时重复利用着色信息。计算 $\boldsymbol{M}^{-1}\boldsymbol{f}$ 的过程如下:

(1)调用算法 4 求解稀疏下三角方程 $\bar{\boldsymbol{L}}\,\boldsymbol{g}_1=\boldsymbol{f}$,得到向量 $\boldsymbol{g}_1$;

(2) 计算对角矩阵 $\boldsymbol{E}$ 与向量 $\boldsymbol{g}_1$ 的乘积 $\boldsymbol{E}\boldsymbol{g}_1$;

(3)调用算法 4 求解稀疏上三角方程 $\bar{\boldsymbol{U}}\boldsymbol{g}=\boldsymbol{E}\,\boldsymbol{g}_1$ 得到向量 $\boldsymbol{g}$,$\boldsymbol{g}$ 即为所求。

2.3 无全局同步并行的算法

2.3.1 全局同步

利用层次信息进行并行需要在层与层之间添加同步点,在上层中的行未计算完成前不能开始下层的计算,这样的全局同步可能使一些本来可以开始的任务延迟开始,导致计算时间变长。以求解下三角方程 $\boldsymbol{L}_1\boldsymbol{x}=\boldsymbol{b}$ 为例,其中系数矩阵 $\boldsymbol{L}_1$ 如式(9)所示,其运行时长大致如图 4 所示。

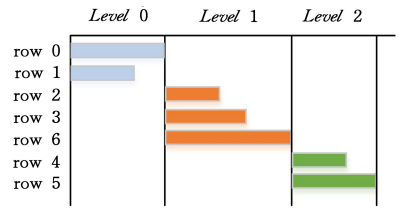


图 4 利用层次信息并行的 $\boldsymbol{L}_1\boldsymbol{x}=\boldsymbol{b}$ 求解时长

Fig. 4 Time elapsed of solving $\boldsymbol{L}_1\boldsymbol{x}=\boldsymbol{b}$ by using levels

显然,下层的一些节点并不依赖于上层中的全部节点,比如 row 2 只依赖于 row 1 而不依赖于 row 0,因此不必等待 row 0 计算完成。考查每个节点真正依赖的节点而不是盲目信任层次信息,可以得到图 5 所示的运行时长图,每行在其所有前驱完成时就开始计算,可以显著减少等待时长。

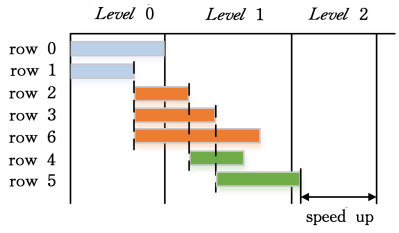


图 5 无全局同步下 $\boldsymbol{L}_1\boldsymbol{x}=\boldsymbol{b}$ 的求解时长

Fig. 5 Time elapsed of solving $\boldsymbol{L}_1\boldsymbol{x}=\boldsymbol{b}$ by synchronization-free scheme

在图 5 所述的思路中,虽然没有了全局同步,但必须为每行设置一个信号量来表示是否计算完成或者是否能够开始计算,用于前驱节点和后继节点之间交流信息。在 GPU 并行中,这个交流信息的开销很难为计算所掩盖;而且由于 GPU 的架构特殊,相邻 32 个线程必须执行相同的指令,很容易造成计算资源的浪费,因此不一定比使用层次信息并行求解快速,数值实验一节将给出对比结果。因此,在实践中是否使用层次信息可以根据前几次的求解时间对比选择。

2.3.2 构造

在无全局同步的算法[22,24]中,需要为每行设置一个信号量用于节点之间交流信息,这个信号量表示本行是否完成计算;后继节点使用本行结果之前必须检查本行的信号量,如果本行未完成计算则必须等待。DILU 预处理子中对角矩阵 $\boldsymbol{E}$ 的计算过程可以用算法 5 描述,其中数组 *ready* 的每个元素对应矩阵的一行,用于交流完成情况。与算法 3 类似,这里也可以先计算 $\boldsymbol{L}$ 和 $\boldsymbol{U}^{\mathrm{T}}$ 的 Hadamard 积 $\boldsymbol{K}$,然后在第 4 行和第 8 行使用 $\boldsymbol{K}$ 进行计算。

算法 5 无全局同步的并行 DILU 构造算法

输入:n×n 严格下三角矩阵 **L**;n×n 严格上三角矩阵 **U**;n×n 对角矩阵 **D**

输出:n×n 对角矩阵 **E**

1. Initialize ready={0,…,0}, whose length is n.
2. parallel for i=0 to n−1 do
3. sum=0
4. for j;j≠i and **L**(i,j)≠0 and **U**(j,i)≠0 do
5. while ready[j]≠1 do
6. busy wait
7. end while

```
8.      sum=sum+L(i,j) * U(j,i) / E(j)
9.    end for
10.  E(i,i)=D(i,i)-sum
11.  ready[i]=1
12. end for
```

算法 5 的策略和图 5 所用策略有所不同，算法 5 采取更精细的控制，仅在需要的某个结果未被计算出来时才等待，而不是等待所有前驱都计算完成才开始计算。值得注意的是，对数组 *ready* 的访问，对 *ready* 的每个元素，最多一个线程写（第 11 行），若干线程读（第 5 行），显然不会造成数据竞争。但若线程内对同一内存位置进行多次读取，且未改变此内存的值，则多次读取可能被编译器优化成一次，这时可以使用 volatile 关键字告知编译器此内存可能因本线程以外的原因而改变，以阻止编译器进行优化。

2.3.3　预处理

预处理的过程主要为求解稀疏三角方程组，本节介绍无全局同步的稀疏三角方程求解算法。与算法 5 类似，为每行设置一个信号量用于表示此行是否完成求解，后继行在使用本行结果之前必须检查对应信号量，必须是完成状态才能使用。无全局同步的稀疏上三角方程并行求解算法如算法 6 所示。稀疏下三角方程的求解稍有不同，在串行算法下，上三角方程使用回代法，而下三角方程使用前代法，因此为保证在线程数量无法覆盖所有行的情况下不发生死锁，上三角方程应当从下向上计算，而下三角方程应当从上向下计算，区别仅在于算法 6 的第 2 行 for 循环范围，下三角方程求解时为“$i=0$ to $n-1$”。

算法 6　无全局同步的稀疏上三角方程并行求解算法[23]

```
输入：n×n 下三角矩阵 U；n×1 右侧向量 b
输出：n×1 结果向量 x
1. Initialize ready={0,…,0}, whose length is n.
2. parallel for i=n-1 to 0 do // 从下向上计算
3.    sum=0
4.    for j:j≠i and U(i,j)≠0 do
5.      while ready[j]≠1 do
6.        busy wait
7.      end while
8.      sum=sum+U(i,j) * x(j)
9.    end for
10.     x(i)=(b(i)-sum) / U(i,i)
11.     ready[i]=1
12. end for
```

2.3.4　无全局同步的着色算法

无全局同步并行策略还可以用于三角矩阵着色算法。在前文中，计算三角矩阵的层次信息时，每一轮都需要大量遍历矩阵的非零元素，这与三角方程分层并行求解是不一样的。着色算法的效率一般低于三角方程的求解，因此即使选择了使用层次信息做 DILU 预处理，利用无全局同步的着色算法计算层次信息也是一个不错的选择。对于下三角矩阵，无全局同步的着色算法的计算过程如算法 7 所示；而上三角矩阵与下三角矩阵的区别仅在于前者需要从下向上计算。

算法 7　无全局同步的稀疏下三角矩阵着色算法

```
输入：n×n 下三角矩阵 L
输出：整数数组 levels，levels[i]即为顶点 i 所在的层数；整数 levcnt，
      表示层次总数
1. Initialize ready={0,…,0}, whose length is n.
2. parallel for i=0 to n-1do // 从上向下计算
3.      left_max=-1
4.    for j:j≠i and L(i,j)≠0 do
5.      while ready[j]≠1 do
6.        busy wait
7.      end while
8.      left_max=Max(left_max,levels[j])
9.    end for
10.     levels[i]=left_max+1
11.     ready[i]=1
12. end for
13. levcnt=Max(levels)+1
```

2.3.5　小结

无全局同步的并行 DILU 预处理技术同样分为构造和预处理两个阶段，相比使用层次信息的算法，无全局同步的算法无须计算层次信息，流程更加简洁。构造阶段的流程为：

（1）取出系数矩阵 $\boldsymbol{A}$ 的严格下三角部分作为 $\boldsymbol{L}$，严格上三角部分作为 $\boldsymbol{U}$，对角部分作为 $\boldsymbol{D}$；

（2）调用算法 5 计算对角矩阵 $\boldsymbol{E}$；

（3）计算 $\bar{\boldsymbol{L}}=\boldsymbol{E}+\boldsymbol{L}$ 和 $\bar{\boldsymbol{U}}=\boldsymbol{E}+\boldsymbol{U}$。

预处理阶段主要为反复求解 $\boldsymbol{M}^{-1}\boldsymbol{f}$，求解流程为：

（1）求解稀疏下三角方程 $\bar{\boldsymbol{L}}\,\boldsymbol{g}_1=\boldsymbol{f}$ 得到向量 $\boldsymbol{g}_1$（算法类似于算法 6）；

（2）计算对角矩阵 $\boldsymbol{E}$ 与向量 $\boldsymbol{g}_1$ 的乘积 $\boldsymbol{E}\,\boldsymbol{g}_1$；

（3）调用算法 6 求解稀疏上三角方程 $\bar{\boldsymbol{U}}\boldsymbol{g}=\boldsymbol{E}\boldsymbol{g}_1$ 得到向量 $\boldsymbol{g}$，$\boldsymbol{g}$ 即为所求。

由于 GPU 硬件构架的特殊性，无全局同步的并行算法并不一定比使用层次信息的并行算法效率高。显然，算法 4 和算法 6 在 GPU 上的执行过程基本只受系数矩阵的影响，在求解同一个方程时，以不同的右侧向量多次调用算法 4 和算法 6，运行时间的差别很小。因此在实践中，为了选择较好的并行策略，可以在前几次迭代中的预处理实施时轮流使用两种策略并对比效率，然后在后续迭代过程中使用效率较高的一个。

3　CUDA 实现

本节介绍一些在 CUDA 平台实现本文算法的关键技术。限于篇幅，本节挑选比较有代表性的算法 2 和算法 7 进行实现，其他算法类似。

首先应当明确稀疏矩阵如何存储，常见的存储格式有 COO，CSR 和 DIA 等，不同的存储格式在各种计算的性能上各有优劣，本节基于 CSR 格式实现。CSR 格式采用 3 个数组表示一个稀疏矩阵，由 COO 格式压缩行得到。仍然以矩阵 $\boldsymbol{L}_1$ 为例，采用 CSR 存储格式表示为（IA，JA，AA）3 个数组。

IA: 0 1 2 4 6 9 12 14

JA: 0 1 1 2 1 3 0 2 4 2 3 5 1 6

AA: 0 1 1 2 1 3 1 1 4 1 1 5 1 6

其第 i 行的非零元素值为: $AA[IA[i],\cdots,IA[i+1]-1]$, 所在的列为: $JA[IA[i],\cdots,IA[i+1]-1]$。在 CSR 格式下, 访问矩阵的一行比较容易, 因此算法应尽量按行访问矩阵, 如果确实需要按列访问矩阵, 则可以将矩阵转置。稀疏矩阵转置是一个并行排序问题。

下面仅给出算法 2 第 1－3 行对应的 CUDA 代码。

```
1. __global__ void ColorTriangularKernel(
2.   int * levels, const int lev,
3.   const int * IA, const int * JA, const int N
4. ) {
5.   int i=threadIdx.x+blockIdx.x * blockDim.x;
6.   for (;i < N;i+=blockDim.x * gridDim.x) {
7.     if (levels[i] ! =-1) continue;
8.     bool can_be_colored=true;
9.     for (int k=IA[i];k < IA[i+1];++k) {
10.       if (JA[k]==i) continue;
11. int levj=levels[JA[k]];
12. if (levj==-1 || levj==lev) {
13.     can_be_colored=false;
14.     break;
15.     }
16.   }
17.     if (can_be_colored) levels[i]=lev;
18.     }
19. }
20. Fill(levels,-1);
21. for (lev=0;ThereIsNegativeIn(levels);lev++) {
22.     ColorTriangularKernel<<<blocks,threads>>>(
23.       levels,lev,IA,JA,N);
24. }
25. levcnt=lev.
```

第 1－19 行为核函数, 在 GPU 上执行; 第 20－25 行为控制逻辑, 在 CPU 上执行。ThereIsNegativeIn 函数用于检查数组 *levels* 中是否还有负数, 如果有, 则未着色完毕, 否则着色结束, 这个函数可以使用并行归约实现。

算法 4 的实现与此类似, 在遍历所有层次的循环中启动核函数, 每个核函数处理一层中的行。由于 CUDA 的核函数在 GPU 上异步执行, 因此启动核函数的开销可以被计算任务覆盖。

在 GPU 上实现算法 7 较为困难, 如果实现时不仔细考虑条件分支的执行情况, 程序极容易陷入死锁状态。这是因为 CUDA 采用 SIMT (Single-Instruction, Multiple-Thread)并行架构。在 CUDA 中, 相邻 32 个线程被称为线程束(Warp), 它们在不同的数据上执行完全相同的指令, 当遇到条件分支(如 if 语句或者 for 循环等)时, 如果线程束内的线程对条件的计算结果不同, GPU 将依次禁用不满足条件的线程, 先后走过所有分支, 这个现象被称为线程束分化(Warp Divergence)。发生线程束分化时, 如果先执行的分支依赖于其他分支, 则会发生死锁, 这种死锁难以被发现, 需要在编码时仔细考虑。为避免死锁, 在实现算法 7 时应当尽量减少条件分支, 简化逻辑, 以下为一个可供参考的实现。

```
1. __global__ void LowerColorKernel(
2.   int * levels, const int * IA, const int * JA, const int N,
3.   volatile int * ready // 使用 volatile 阻止编译器优化
4. ) {
5.   const int i=threadIdx.x+blockIdx.x * blockDim.x;
6.   if (i < N) {
7.     int left_max=-1;
8.     int j=IA[i];
9.     while (j < IA[i+1]) {
10.       const int col=JA[j];
11.       while (ready[col]==1) { //最多到对角元素
12.         left_max=max(left_max,levels[col]);
13.         ++j;
14.       }
15.       if (col==i) {//对角元素为哨兵
16.         levels[i]=left_max+1;
17.         ready[i]=1;
18.         ++j;
19.       }
20.     }
21.   }
22. }
23. cudaMemset(ready,0,N * sizeof(int));
24. LowerColorKernel<<<blocks,threads>>>(
25.   levels,IA,JA,N,ready
26. );// 线程数量需要覆盖所有行
27. levcnt=Max(levels,N)+1.
```

在算法 7 的 CUDA 实现中, 需要仔细考虑线程束分化的情况, 主要的分化出现在第 11 行的 while 循环和第 15 行的 if 语句。某个线程在第 11 行遇到某个前驱未计算完毕时则提前退出循环, 且被禁用至线程束内其他 31 个线程都退出循环, 然后一起进入第 15 行的条件分支。在第 15 行的条件分支中, 未遍历完前驱的线程将被禁用, 而遍历完前驱的线程可以执行第 16－18 行的代码, 计算本行的层次并标记为计算结束。第 18 行的 j 自增是为了在下一步, 即第 9 行退出循环。第 18 行不能使用 break 语句跳出循环, 因为 break 将引起新的线程束分化, 导致死锁。最后, 第 27 行的 Max 函数为求数组的最大值, 可以通过并行归约实现。

算法 7 也可以实现为一个线程束计算一行, 这样可以大大减少线程束分化的出现次数, 但许多稀疏矩阵每行的非零元素非常少, 使用 32 个线程计算一行会导致大量线程空载, 白白浪费硬件资源, 这也正是文献[24]提出线程级无全局同步的并行三角方程求解算法的动机之一。

4 数值实验

本节介绍关于 DILU 预处理技术的一些数值实验结果, 主要有: DILU 和 ILU 构造阶段耗时对比、DILU 和 ILU 预处理效果对比、各矩阵层次信息统计和两种并行策略下预处理的效率对比, 以及 GPU 和 CPU 运行的 DILU 预处理技术的性能对比。

本节测试平台的 CPU 为两个 6 核的 E5-2620@2.0 GHz，GPU 为 Tesla K20m，操作系统为 Centos 6.2，编译器使用 gcc 4.8.5，CUDA 的版本为 6.5，程序使用 C++和 CUDA 进行编写，启用 O3 优化。

本节从 SuitSparse Matrix Collection（sparse.tamu.edu）选取了一些方程进行测试，这些方程的信息如表 1 所列。

表 1　用于测试的方程

Table 1　Information of matrices used in test

No.	Matrix	M，N	NNZ	Kind
1	apache2	715 176	4 817 870	Structural Problem
2	ASIC_320ks	321 671	1 316 085	Circuit Simulation
3	atmosmodd	1 270 432	8 814 880	Atmospheric Models
4	cage13	445 315	7 479 343	DNA Electrophoresis
5	G3_circuit	1 585 478	7 660 826	Circuit Simulation
6	parabolic_fem	525 825	3 674 625	CFD

4.1　DILU 和 ILU 的性能对比

DILU 和 ILU 预处理技术都分为构造阶段和预处理实施阶段。在预处理实施阶段，主要求解稀疏三角方程，对于同一个方程，DILU 和 ILU 要求解的三角方程零模型完全相同，因此耗时差别不大。本节主要对比 DILU 和 ILU 在构造阶段的耗时，以及它们对迭代法的加速效果。

DILU 和 ILU 对系数矩阵没有要求，因此适用于对目标为一般方程组的迭代法进行加速，这类迭代法有广义极小残量法、稳定双共轭梯度法等。本节采取稳定双共轭梯度法作为基本的迭代方法，分别使用 DILU 和 ILU 预处理技术进行加速，以对比它们的性能和效率。预处理稳定双共轭梯度法如算法 1 所示。

DILU 的实现如前文所述，含有利用层次信息并行和无全局同步并行两种策略的实现，其中着色算法采用算法 7 而非算法 2。ILU 预处理子采用 CUDA 工具包自带的 cusparse 库进行构造[27]，对应的函数为 cusparseDcsrilu02，这个函数也含有利用层次信息并行和不使用层次信息两种策略，本节将对两种策略进行对比测试。

首先，我们测试了构造这些矩阵的 DILU 预处理子和 ILU 预处理子的耗时，如表 2 所列。对于 DILU 预处理子的构造而言，使用无全局同步的并行算法更具有优势，但个别方程使用层次信息构造耗时更短。对 ILU 预处理子的构造，两种策略都没有明显的优势，在实践中可以轮流使用并选择性能更好的那一个。DILU 和 ILU 的构造时间相比，DILU 明显优于 ILU，并且 DILU 的耗时更加稳定，没有出现过较大的值。在预处理实施阶段，ILU 的主要计算任务为求解一个下三角方程和一个上三角方程，而 DILU 需要多求一个对角矩阵和向量的积。由于 DILU 和 ILU 中三角方程的系数矩阵零模型完全相同，并且在 GPU 上对角矩阵乘向量的计算速度非常快，因此在预处理实施阶段上，相同并行策略下的 DILU 和 ILU 预处理的耗时基本没有区别，本文不再进行对比。

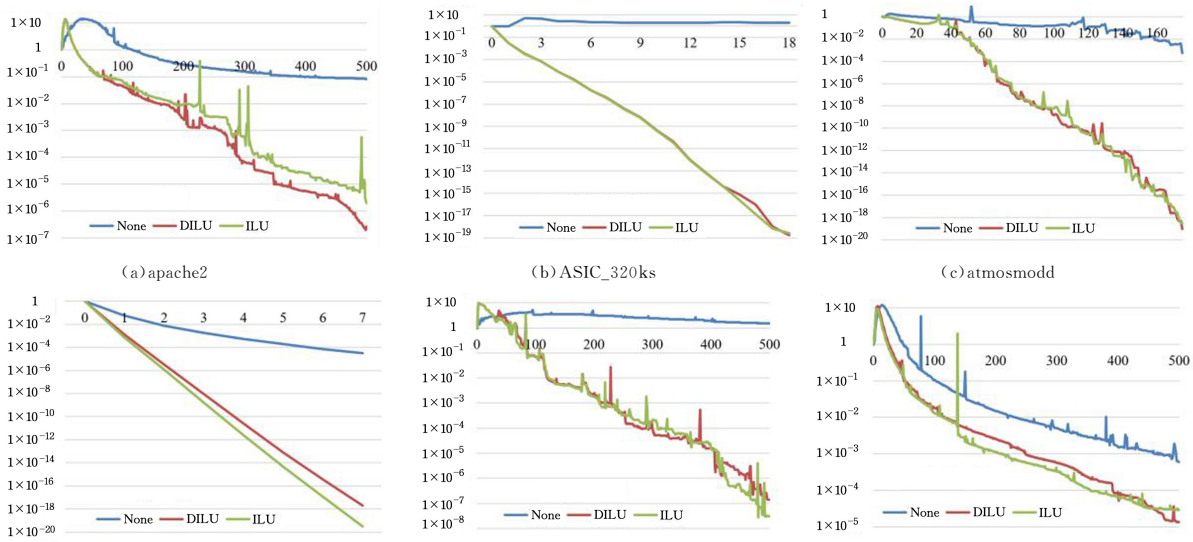
表 2　预处理子构造耗时

Table 2　Time elapsed of preconditioner construction

（单位：ms）

No.	DILU		ILU	
	Use Levels	*No Levels*	*Use Levels*	*No Levels*
1	74.1647	84.9158	96.5731	372.74700
2	34.5870	28.4623	31.7350	25.7219
3	87.8330	106.7720	125.0960	377.5720
4	62.0161	55.4290	71.1740	59.6290
5	134.9230	130.0960	163.3960	715.1190
6	34.7684	29.4419	34.0866	28.8091

分别使用 DILU 预处理和 ILU 预处理的稳定双共轭梯度法求解以表 1 中的矩阵为系数矩阵的方程 $\boldsymbol{Ax}=\boldsymbol{b}$，方程的右侧向量 $\boldsymbol{b}$ 的每个元素都为 0～1 之间的随机数，迭代的初始值为 $\boldsymbol{x}_0=[0,0,\cdots,0]^{\mathrm{T}}$；统计每步迭代之后的残量 $\boldsymbol{r}_k=\boldsymbol{b}-\boldsymbol{Ax}_k$ 的二 JP2]范数，将它们与初始残量 $\boldsymbol{r}_0=\boldsymbol{b}-\boldsymbol{Ax}_0$ 的二范数作对比，可以测得 $\|\boldsymbol{r}_k\|/\|\boldsymbol{r}_0\|$ 在迭代过程中的下降情况，如图 6 所示。



(a) apache2　(b) ASIC_320ks　(c) atmosmodd

(d) cage13　(e) G3_circuit　(f) parabolic_fem

注：横坐标为迭代步数，纵坐标为 $\|\boldsymbol{r}_k\|/\|\boldsymbol{r}_0\|$，且纵坐标采用对数坐标；None，DILU 和 ILU 3 条曲线分别为无预处理、DILU 预处理和 ILU 预处理的 BiCG-Stab 方法求解的情况

图 6　$\|\boldsymbol{r}_k\|/\|\boldsymbol{r}_0\|$ 随迭代步数增加的下降曲线

Fig. 6　Curve of $\|\boldsymbol{r}_k\|/\|\boldsymbol{r}_0\|$ in iterations

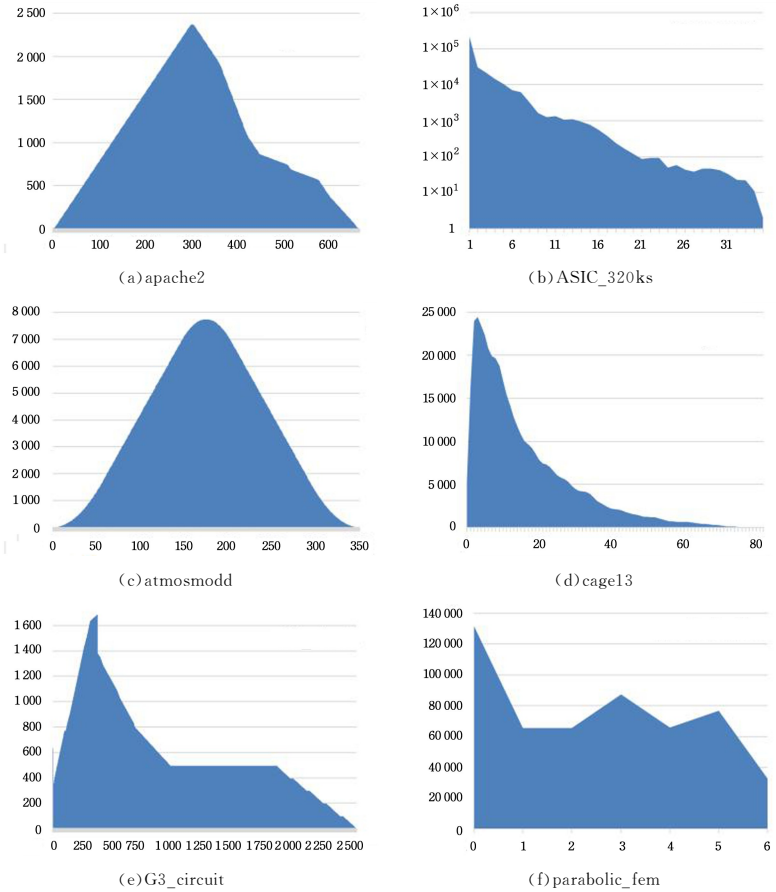
在图 6 的残量下降曲线中，可以明显观察到 DILU 和 ILU 预处理都可显著地对迭代法进行加速，两者的曲线相差较小，对迭代法的加速效果接近，但 DILU 预处理下的曲线相对光滑一些，数值稳定性较好。

在实验中，也会遇到一些使用 DILU 预处理效果不佳的方程，例如同样选自 SuitSparse Matrix Collection（sparse.tamu.edu）的方程 offshore，无预处理和使用 DILU 预处理的 BiCGStab 都未能收敛，但使用 ILU 预处理效果非常好，因此在选择预处理技术时应当根据实际方程的求解情况综合考虑。

4.2 两种并行策略的对比

本文研究了利用层次信息的并行策略和无全局同步的并行策略两种策略下并行DILU预处理技术的实现，两种并行策略下构造阶段的效率见表 2，本节主要关注实施预处理时两种并行策略的效率。

本节测试时使用的矩阵不全是对称的，但它们的零模型都是对称的，因此在 DILU 和 ILU 预处理子中，下三角矩阵和上三角矩阵的零模型也是对称的，它们的着色结果大同小异，这里仅展示下三角矩阵着色的结果。从 DILU 和 ILU 的构造方法可知，预处理子中的下三角矩阵和方程系数矩阵的下三角部分的零模型完全相同，它们的着色结果完全相同。图 7 给出了 DILU 和 ILU 预处理子中下三角矩阵着色后各层中的行数分布，亦是系数矩阵下三角部分着色后各层次中行数的分布。



注：横坐标为层次，纵坐标为行数，且矩阵 ASIC_320ks 的纵坐标采用了对数坐标

图 7 下三角矩阵各层的行数分布

Fig. 7 Distribution of rows into levels of lower triangular matrix

从图 7 可以看出，矩阵着色后，各层内的行数分布十分不均匀，尤其是某些层次内的行数非常少，不利于并行。在求解三角方程时，如果利用层次信息并行，则需要在 for 循环中启动核函数依次处理各层，启动核函数存在一个固有开销，如果任务规模不够大，运行时间过短，则无法掩盖启动核函数消耗的时间，造成计算任务之外的开销。

如果使用无全局同步的并行算法，可以避免重复启动核函数的开销，但由于线程束分化的存在，性能不一定比使用层次信息更优。我们分别使用两种并行策略的 DILU 预处理 BiCGStab 方法来求解表 1 中的方程，所用时长如表 3 所列。

表 3 两种并行策略下 DILU 预处理 BiCGStab 的耗时

Table 3 Time elapsed of DILU BiCGStab under two parallel schemes

（单位：s）

No.	*Iterations*	*None*	*Coloring*	*Use Level*	*No Level*
1	500	1.914410	0.082271	15.390600	158.273000
2	18	0.037312	0.011654	0.156014	0.278192
3	175	1.185750	0.086148	6.187200	54.319 600
4	7	0.028656	0.014765	0.120256	0.261664
5	500	4.072180	0.114982	52.062200	234.352000
6	500	1.607330	0.008473	3.702190	3.596050

注：*Iterations* 列是各个方程的迭代步数，*None* 列是无预处理的 BiCGStab 方法花费的时间，*Coloring* 列是着色花费的时间，*Use Level* 列是使用层次信息并行的预处理 BiCGStab 花费的时间，*No Level* 列是无全局同步的预处理 BiCGStab 花费的时间

从表 3 可知，无全局同步的并行算法虽然看似更具优势，但在实践中表现不佳，除个别方程比使用层次信息并行更优以外，在大多数方程上的运行时间都较长。因此在实践中，可以根据前几次迭代的表现选择并行方案。相比无预处理的 BiCGStab 在相同迭代步数下的运行时间，无论是分层并行策略还是无全局同步的并行策略，预处理 BiCGStab 的运行时间都要长许多，因此求解稀疏三角方程是预处理 BiCGStab 方法的性能瓶颈，希望在未来可以出现更好的算法。

4.3　GPU 和 CPU 实现的对比

本节对比 GPU 和 CPU 上 DILU 预处理迭代法的性能。在 CPU 的实现中，除预处理子的构造和实施预处理两个操作外，其他他操作如矩阵乘向量使用 intel 的 TBB 框架并行，以 12 个线程运行，刚好跑满 CPU 的核心。

首先是 DILU 预处理子的构造。表 1 中的方程在 GPU 和 CPU 上构造 DILU 预处理子的耗时如表 4 所列。从表 4 可知，无论是按层次并行还是无全局同步的并行策略，GPU 上的构造速度都明显快于 CPU。

表 4　GPU 和 CPU 上构造 DILU 预处理子的耗时

Table 4　Time elapsed of DILU preconditioner construction on GPU and CPU

（单位：ms）

No.	GPU		CPU
	Use Levels	*No Levels*	
1	74.1647	84.9158	122.2930
2	34.5870	28.4623	56.4922
3	87.8330	106.7720	214.3640
4	62.0161	55.4290	201.4720
5	134.9230	130.0960	214.4830
6	34.7684	29.4419	106.6000

其次是 DILU 预处理在迭代法中的性能表现。表 1 中的方程分别在 GPU 和 CPU 上利用 DILU 预处理的稳定双共轭梯度求解时的耗时如表 5 所列。从表 5 可知，无全局同步的 GPU 并行的迭代法可能会比 CPU 执行更慢，但按层次并行的方法始终比 CPU 快，因此 GPU 并行的 DILU 预处理具有更好的性能。在 OpenFOAM 中，进程内部使用 CPU 串行执行，如果更换成 GPU 并行执行，速度将会有明显提升。

表 5　GPU 和 CPU 上求解方程的耗时

Table 5　Time elapsed of DILU BiCGStab on GPU and CPU

（单位：s）

No.	Iterations	GPU		CPU
		Use Levels	*No Levels*	
1	500	15.472871	158.273000	37.998400
2	18	0.167668	0.278192	0.462310
3	175	6.273348	54.319600	23.716200
4	7	0.135021	0.261664	0.393182
5	500	52.177182	234.352000	79.348000
6	500	3.710663	3.596050	21.089400

结束语　本文讨论了 DILU 预处理技术在 GPU 上的实现，研究了两种并行策略，即利用层次信息并行求解策略和无全局同步的并行求解策略，并对两种并行策略下的 DILU 预处理的性能作了对比。两种策略无绝对的好坏，分层并行策略需要全局同步，且需要做额外的着色工作；而无全局同步的策略可以避免层与层之间的同步开销。但是从数值实验结果来看，分层并行策略在许多方程中都有不错的表现。在实践中，可以综合考虑两种策略，选择较优的一种。

DILU 和 ILU 预处理都能对迭代方法起到很好的加速效果，在很多方程上，DILU 的数值稳定性优于 ILU，使用 ILU 做预处理的程序可以考虑提供 DILU 预处理技术。数值实验结果表明，GPU 并行的 DILU 预处理技术比 CPU 执行的拥有更好的性能，可以考虑将现有的 CPU 运行的软件移植到 GPU 平台以获得更好的性能。DILU 预处理也有缺点，其带来了较高的计算开销，尤其是在 GPU 上，三角方程的求解效率远不如稀疏矩阵乘向量等矩阵和向量运算的效率，成为了预处理迭代方法的性能瓶颈，未来可以重点研究稀疏三角方程的并行求解算法。

参 考 文 献

[1] SAAD Y. Iterative Methods for Sparse Linear Systems [M/OL]. SIAM, 2003. https://www-users. cse. umn. edu/~saad/itermethbook_2nded. pdf.

[2] GUTKNECHT M H. A brief introduction to Krylov space methods for solving linear systems[M]// Frontiers of Computational Science. Berlin: Springer, 2007: 53-62.

[3] LANGR D, TVRDIK P. Evaluation Criteria for Sparse Matrix Storage Formats[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(2): 428-440.

[4] DAVIS T A, RAJAMANICKAM S, SID-LAKHDAR W M. A Survey of Direct Methods for Sparse Linear Systems[J]. Acta Numerica, 2016, 25: 383-566.

[5] HACKBUSCH W. Iterative Solution of Large Sparse Systems of Equations (2nd ed)[M]. Berlin: Springer, 2016.

[6] FILIPPONE S, CARDELLINI V, BARBIERI D, et al. Sparse Matrix-vector Multiplication on GPGPUs[J]. ACM Transactions on Mathematical Software (TOMS), 2017, 43(4): 30.

[7] STEINBERGER M, DERLERY A, ZAYER R, et al. How Naive is Naive SpMV on the GPU? [C]// 2016 IEEE High Performance Extreme Computing Conference(HPEC). IEEE, 2016: 1-8.

[8] GAO J, QI P, HE G. Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU[J]. Mathematical Problems in Engineering, 2016, 2016: 1-14.

[9] BELL N, GARLAND M. Efficient sparse matrix-vector multiplication on CUDA: Nvidia Technical Report: NVR-2008-004[R]. Nvidia Corporation, 2008.

[10] BARRETT R, BERRY M W, CHAN T F, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods[M]. SIAM Other Titles in Applied Mathematics, 1994.

[11] BEHRENS T. OpenFOAM's basic solvers for linear systems of equations[J/OL]. Chalmers, Department of Applied Mechanics, 2009, 18(2). http://www. tfd. chalmers. se/~hani/kurser/os_cfd_2008/timberhrens/tibeh-report-fin. pdf.

[12] OpenFOAM Project[EB/OL]. https://openfoam. com/.

[13] OWENS J D, HOUSTON M, LUEBKE D, et al. GPU computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.

[14] MEIJERINK J A, VAN DER VORST H A. An Iterative Solu-

tion Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix[J]. Mathematics of Computation, 1977, 31(137): 148-162.

[15] KERSHAW D S. The incomplete Cholesky—conjugate gradient method for the iterative solution of systems of linear equations [J]. Journal of Computational Physics, 1978, 26(1): 43-65.

[16] SAAD Y. ILUT: A dual threshold incomplete LU factorization [J]. Numerical Linear Algebra with Applications, 1994, 1(4): 387-402.

[17] POMMERELL C. Solution of large unsymmetric systems of linear equations[D]. ETH Zürich: Swiss Federal Institute of Technology Zurich, 1992.

[18] ANDERSON E, SAAD Y. Solving Sparse Triangular Linear System on Parallel Computers[J]. International Journal of High Speed Computing, 1989, 1(1): 73-95.

[19] NAUMOV M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU: NVIDIA Technical Report: NVR-2011-001[R]. 2011.

[20] LI R, SAAD Y. GPU-accelerated preconditioned iterative linear solvers[J]. The Journal of Supercomputing, 2013, 63(2): 443-466.

[21] NAUMOV M, CASTONGUAY P, COHEN J. Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU: NVIDIA Technical Report: NVR-2015-001[R]. 2015.

[22] LIU W, LI A, HOGG J, et al. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves[C]// European Conference on Parallel Processing. Cham: Springer, 2016: 617-630.

[23] LI R. On parallel solution of sparse triangular linear systems in CUDA[J]. arXiv: 1710.04985, 2017.

[24] SU J, ZHANG F, LIU W, et al. CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs [C]// 49th International Conference on Parallel Processing—ICPP. 2020.

[25] XIE C, CHEN J, FIROZ J S, et al. Fast and Scalable Sparse Triangular Solver for Multi-GPU Based HPC Architectures[J]. arXiv:. 06959, 2020.

[26] ANZT H, RIBIZEL T, FLEGAR G, et al. ParILUT—A Parallel Threshold ILU for GPUs[C]// 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2019: 231-241.

[27] CHEN Y, TIAN X, LIU H, et al. Parallel ILU preconditioners in GPU computation[J]. Soft Computing, 2018, 22(24): 8187-8205.

[28] NAUMOV M. Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS[J/OL]. Nvidia white paper, 2011: 1-16. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.230.8934&rep=rep1&type=pdf.

[29] BNA S, SPISSOA I, OLESENB M, et al. PETSc4FOAM: A Library to plug-in PETSc into the OpenFOAM Framework[J/OL]. PRACE White paper, 2020. https://prace-ri.eu/wp-content/uploads/WP294-PETSc4FOAM-A-Library-to-plug-in-PETSc-into-the-OpenFOAM-Framework.pdf.

[30] Paralution project[EB/OL]. http://www.paralution.com.

[31] 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.

[32] VAN DER VORST H A. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems[J]. SIAM Journal on Scientific and Statistical Computing, 1992, 13(2): 631-644.

[33] JENSEN T R, TOFT B. Graph Coloring Problems [M/OL]. 1995. https://doi.org/10.3929/ethz-a-000669614.

[34] SALTZ J H. Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors[J]. SIAM Journal on Scientific and Statistical Computing. 1990, 11(1): 123-144.

[35] KNUTH D E. The art of computer programming, volume 3: (2nd ed.) sorting and searching[M]. Addison Wesley Longman Publishing Co., Inc., 1998, 17(4): 324.

[36] CORMEN T H, LEISERSON C E, RIVEST R L, et al. Introduction to algorithms second edition[M]. MIT Press, 2001.

[37] HORN R A, JOHNSON C R. The Hadamard product[M]// Topics in Matrix Analysis: Cambridge University Press, 1991: 298-381.

WANG Jin, born in 1995, postgraduate. His main research interests include parallel computing and iterative methods for linear systems.

LIU Jiang, born in 1979, Ph.D, associate professor, is a member of China Computer Federation. His main research interests include computability theory, formal methods and computer algorithms.

(责任编辑:柯颖)