Pure Mathematics 理论数学, 2025, 15(4), 84-90
Published Online April 2025 in Hans. https://www.hanspub.org/journal/pm
https://doi.org/10.12677/pm.2025.154112

Hans汉斯

基于LU分解并行求解稠密线性方程组

王　郎

成都理工大学数学科学学院，四川　成都

收稿日期：2025年2月26日；录用日期：2025年3月17日；发布日期：2025年4月9日

摘　要

快速求解线性方程组是电磁场、流体力学、结构力学等许多工程问题的核心，LU分解是求解稠密线性方程组最常用的方法之一。该文详细描述了LU分解的原理和过程，基于此过程的特点，在CUDA (Compute Unified Device Architecture)并行环境下，提出一种的并行计算方法，该方法可以将计算任务划分为若干小问题进行求解,各部分求解完成后再通过前向回代和后向回代，即可获得最终解。最后通过实验证明，该方法能够充分利用GPU，有效地减少了数据间的通信开销，从而加快了求解速度，提高了计算效率。

关键词

LU分解，并行计算，稠密线性方程组，CUDA

Parallel LU Decomposition for Solving Dense Linear Systems

Lang Wang

School of Mathematical Sciences, Chengdu University of Technology, Chengdu Sichuan

Received: Feb. 26th, 2025; accepted: Mar. 17th, 2025; published: Apr. 9th, 2025

Abstract

Rapid solving of linear systems is central to engineering challenges in electromagnetics, fluid dynamics, structural mechanics, and more. LU decomposition remains one of the most widely used methods for dense linear systems. This paper elaborates on the principles and process of LU decomposition. Leveraging its algorithmic characteristics, we propose a parallel computation framework on CUDA that decomposes the computational task into smaller sub-problems. After solving these sub-tasks independently, the final solution is reconstructed through forward and backward substitutions. Experimental results demonstrate that this method maximizes GPU utilization, significantly reduces inter-node communication overhead, and achieves accelerated solving speeds with

文章引用: 王郎. 基于 LU 分解并行求解稠密线性方程组[J]. 理论数学, 2025, 15(4): 84-90.
DOI: 10.12677/pm.2025.154112

improved computational efficiency.

Keywords

LU Decomposition, Parallel Computing, Dense Linear Systems, CUDA

1. 引言

随着计算机应用对算力需求的指数级增长及计算场景的多元化发展，处理器经过不断的革新。虽然处理器的计算性能不断提升，但是受纳米级集成电路工艺的物理的限制，单核处理器的主频提升已接近瓶颈，性能优化空间逐渐缩小。在这个背景下，通过多核并行计算架构[1]来提升计算性能，已成为当前计算技术研究的核心方向。

面对高分辨率气候建模、超大规模深度学习模型训练等复杂计算场景，并行计算技术[2]通过多节点任务协同与硬件的加速能力，已经成为突破单核处理器计算性能瓶颈的关键方法。其中，基于 MPI (消息传递接口)的分布式内存并行架构[3]，以及依靠 GPU (图形处理单元)的众核并行加速技术[4]，不仅通过细粒度任务划分与流水线调度实现了计算资源的高效利用，更在异构计算框架下为科学计算与工业智能化提供了高效的解决方案。

在电磁场、流体力学、结构力学等许多工程问题和物理问题都涉及到大规模的线性方程组的求解[5]。大规模稠密线性方程组矩阵的求解成为了计算模拟中的关键步骤[6]。高斯消元法[7]和 LU 分解[8]作为两种经典的直接求解方法，因其较为稳定且易于实现的特点，被广泛应用于线性方程组的求解中。对于稠密矩阵，通常使用 LU 分解，LU 分解即将一个线性方程组的系数矩阵分解为一个单位下三角矩阵和一个上三角矩阵，再利用前向回代和后向回代求出解向量 x，这种方法非常适用于稠密矩阵的情况。基于 MPI 或 GPU 的并行计算技术的应用，为解大规模的线性方程组提供了有效的解决方案。将 LU 分解过程中的计算任务分配给多个处理单元并行执行，可以大大的缩短计算时间并提高资源的利用率。基于 GPU 的并行计算依靠其强大的并行处理能力，在处理大规模稠密线性方程组时，能够大幅提升矩阵分解过程的计算效率。

CUDA (Compute Unified Device Architecture)是由 NVIDIA 公司推出的一种并行计算平台和编程模型[9]，开发者能够利用 GPU 进行并行计算。2006 年发布到现在为止，CUDA 已成为高性能计算领域的一个重要工具，广泛的应用于图像处理、机器学习、科学计算、金融建模等多个领域。因为 GPU 比传统的 CPU 具有更高的并行处理能力，所以在计算大量的数据时，利用 GPU 能够大大提高计算效率。

本文提出一种基于 CUDA 的 LU 分解并行算法，将矩阵划分为动态优化调整的线程块网格，利用共享内存的高速缓存来加速 LU 分解，该设计直接通过 GPU 显存来进行数据的交互，避免 MPI 的各个进程之间的通信延迟，从而提高了线性方程组的求解效率。

2. 稠密线性方程组的并行计算简述

在计算科学与应用数学交叉领域，并行计算作为一种新型的计算模式，它是一种通过同时使用多核处理器、GPU 以及分布式计算集群的等多个计算资源，来加快求解大规模的复杂问题的新型计算方式，

其核心在于将大规模问题分解为可并行的子任务，利用计算资源的空间并行性来加速问题求解的效率。

线性方程组是由多个线性方程所组成的，其目标是求解一组未知数的值，使得这组数满足每一个线性方程，而这类线性方程组的系数矩阵通常为是稠密矩阵，即非零元素占比超过 90%的系数矩阵。

3. LU 分解法原理

计算求解 $Ax=b$ 的线性方程组，假设 A 的顺序主子式都不为零，且本文中仅讨论当 A 为方阵时的情景，那么 A 可以进行 LU 分解，即 $A=LU$，其中 L 为单位下三角矩阵，U 为上三角矩阵，

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & & 0 \\ \vdots & \ddots & \\ l_{n1} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ 0 & & u_{nn} \end{pmatrix}$$

由 LU 分解法可知，先计算 U 的第一行，可得 $u_{1j}=a_{1j}, j=1,2,\cdots,n$；再计算 L 的第一列，可得 $l_{r1}=a_{i1}/a_{11}, i=2,\cdots,n$，依次循环，则计算 U 的第 r 行为 $u_{rj}=a_{rj}-\sum_{k=1}^{r-1} l_{rk}u_{kj}, j=r,r+1,\cdots,n$；接着计算 L 的第 r 列 $l_{ir}=\dfrac{\left(a_{ir}-\sum_{k=1}^{r-1} l_{ik}u_{kr}\right)}{u_{rr}}, i=r+1,\cdots,n$，即 A 的 LU 分解可以按照 U_1 行 L_1 列，U_2 行 L_2 列，…，依次进行，直至分解完成，得到一个单位下三角矩阵 L 和一个上三角矩阵 U。对于线性方程组 $Ax=b$，利用 LU 分解法得到 L 和 U 后，根据 L 和 U 的特性，可利用 $Ly=b, Ux=y$，先向前回代求解 $Ly=b$ 求出 y，再向后回代求解 $Ux=y$，最后求出解向量 x。

4. CUDA 编程模型

在 CUDA 架构中，引入了主机(Host)和设备(Device)这两个概念，其中，CPU 及系统内存作为 Host，负责任务调度、控制流程以及串行计算，而 GPU 及其显存作为 Device，专门用于并行计算和数据处理。在程序执行过程中，CUDA 在单指令多数据(SIMD, Single Instruction Multiple Data)模型的基础上进行了优化，采用单指令多线程(SIMT, Single Instruction Multiple Thread)模型，使得同一指令序列能够生成多个 CUDA 线程，并将一个线程束中的 CUDA 线程映射到 GPU 不同的 CUDA 核心，独立处理不同的数据。相比于 SIMD，SIMT 提供了更高的灵活性，能够自动适应不同的执行宽度。

4.1. 线程模型

CUDA 程序的线程模型由线程块层和线程层两个并行逻辑层组成。在程序执行前，数据需要通过特定接口传输到设备端内存，随后通过核函数启动大量 GPU 线程并行处理数据。在执行过程中，线程块被映射到流式多处理器，线程被映射到流处理器。不同的线程块可以被映射到同一流式多处理器上并行处理，而每个线程块又会被划分为多个线程束，每个线程束由线程块中连续的 32 个线程组成。线程块是 GPU 上最小的并行执行单元，包含多个线程，一个线程块内的所有线程可以在一片共享内存区域内进行数据通信，降低数据访问的延迟。网格是由多个线程块组成的，用来协调这些线程块的执行，可以用来处理更大的计算任务。每个线程块负责处理任务的一部分，从而实现高效的并行。核函数是在 GPU 上运行的 CUDA 函数，需要使用_global_进行声明。CUDA 函数的并行执行逻辑包括粗粒度并行和细粒度并行两个层次。其中，粗粒度并行发生在网格内的线程块之间，即多个线程块可独立并行执行计算任务，提高整体并发性。而细粒度并行则发生在线程块内部的不同线程之间，各线程能够同时执行计算，从而进一步优化计算效率并提升性能。

4.2. 存储模型

CUDA 的内存层次结构包含了多种内存类型，分别是全局内存、共享内存、常量内存和纹理内存。全局内存是 GPU 中最大的内存，但是访问速度慢，容易影响计算性能；共享内存是线程块内的高速缓存，虽然容量较小但是可以实现数据共享，访问速度快，大大提高计算性能。常量内存具有专门的缓存，在所有流式多处理器之间共享，主要用于存储不变的数据，当多个线程访问同一个常量值时，GPU 会利用广播机制，一次读取就能满足所有线程的需求，避免冗余访问。纹理内存具有独立的缓存系统，支持二维、三维访问模式，能够有效提高数据局部性并减少内存延迟。

5. CUDA 并行求解算法设计

在 CUDA 并行求解算法设计中，主要利用 GPU 强大的并行计算能力来加速 LU 分解过程，特别是在矩阵进行 LU 分解时的行列消元过程中利用 GPU 的并行计算能力，最后求出解向量 x。我们将分解任务划分为多个线程块，每个线程块负责矩阵的一个小部分，充分利用 GPU 的并行架构和共享内存的数据共享和高速缓存，来降低内存访问的延迟和突破计算性能瓶颈。为了有效地解决大规模线性方程组，本文提出了一种基于 CUDA 的并行 LU 分解方法。在传统的 LU 分解中，每次行列计算都需要访问前面的计算结果，因此行列计算过程的数据依赖性非常强，将此任务并行化可以大大的减少计算时间。为了提高数值计算的稳定性，我们在 LU 分解中加入了部分选主元技术。在每一步消元之前，我们选择当前列中绝对值最大的元素作为主元，并交换两行元素，以防止计算过程中的数值不稳定问题。不可避免的，当加入部分选主元技术以后，由于寻找主元以及交换两行等操作，会增加通信开销。但是，LU 分解在处理各种不同的矩阵，甚至是在处理接近奇异的矩阵时，部分选主元技术可以使得使用 LU 分解来分解这个矩阵时更加可靠稳定。

5.1. 并行策略

观察 LU 分解过程，采取迭代并行策略：由于 LU 分解对 U 的第 r 行和 L 的第 r 列进行分解时，该行的待分解元素所需数据只有该行主元及左侧所有元素以及待分解元素所在列的上方所有元素，该列的待分解元素所需数据只有该列主元上方所有元素以及待分解元素所在行的左侧所有元素，所以同行和同列的元素进行分解时并无数据依赖，由此可以并行分解 U 的第 r 行和 L 的第 r 列，即先并行分解 U 的第一行和 L 的第一列，再并行分解 U 的第二行和 L 的第二列，依次分解直到完成，过程如图 1 所示。

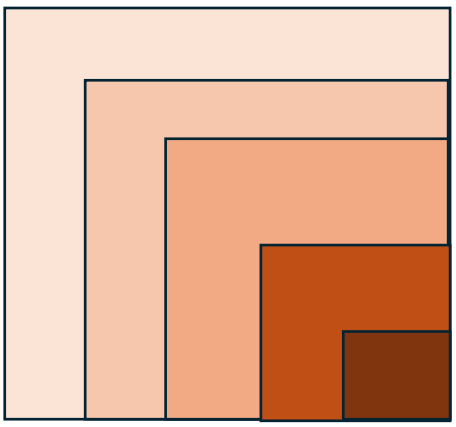

Figure 1. The process of LU decomposition
图 1. LU 分解过程

在计算时，首先将第一步对第一行和第一列并行求解，第二步再对剩余子块并行更新，更新后在继续对第二行和第二列并行求解，求解后再对剩余子块并行更新，直到整个矩阵分解完成后，利用前向回代和后向回代求出解向量。并行 LU 分解中，系数矩阵 A 的分解过程可以通过一个线程计算一个矩阵元素的方式来并行化。在 CUDA 中，我们使用二维网格和线程块的方法，每个线程块处理矩阵的一部分元素，线程块内的线程则负责计算该部分矩阵中的元素，利用共享内存来减少数据通信时间。这种方式利用了 GPU 的并行计算能力，将大规模的矩阵分解任务分解为多个小规模的任务，分别由不同的线程块和线程处理。每个线程块负责处理矩阵的一个子块，线程块内的线程则并行计算该子块中的元素，来实现高效的并行。而且二维网格的结构正好能够与矩阵的二维布局一一映射，使得数据访问更加高效，从而降低内存访问延迟。在本研究中，我们采用行循环划分(row-cyclic partitioning)策略，其中每个线程计算矩阵 A 中的一个元素，以最大化并行度。采用二维线程块(Block)和二维网格(Grid)进行任务分配，每个线程(threadIdx.x, threadIdx.y)计算一个 A[i] [j]。网格大小 gridDim = (N/BlockSize, N/BlockSize)，线程块大小 blockDim = (BlockSize,BlockSize)，使得每个 BlockSize × BlockSize 线程块负责矩阵的一个子块，提高 GPU 计算效率。为了减少全局内存访问，使用共享内存 pivotRow 存储主元行 A [k]，由 threadIdx.y == 0 线程加载，并使用共享内存 pivotElement 存储 A [k] [k]，由 threadIdx.x == 0&&threadIdx.y == 0 线程加载。在每轮 k 迭代中，所有线程首先加载 pivotRow 和 pivotElement，然后执行__syncthreads()确保数据一致，再并行计算 A [i] [j]更新，并在计算后再次同步，以保证数据正确性。

以 8 × 8 的矩阵为例，使用 2 × 2 的线程块，线程块按 4 × 4 大小划分整个矩阵，TB (x, y)代表线程块 blockIdx.x = x，blockIdx.y = y，每个 TB (x, y)负责计算 4 × 4 的矩阵子块。每个 4 × 4 的线程块由 Block (4, 4)组成，每个线程(threadIdx.x, threadIdx.y)计算 A [i] [j]，以第一个线程块 TB (0, 0)中的线程展开，第一步对矩阵第一行和第一列进行并行分解，第二步对剩余的矩阵进行并行更新元素，接着再对矩阵第二行第二列并行分解，再对剩余矩阵进行更新，一直到整个矩阵分解完成，如图 2 所示。

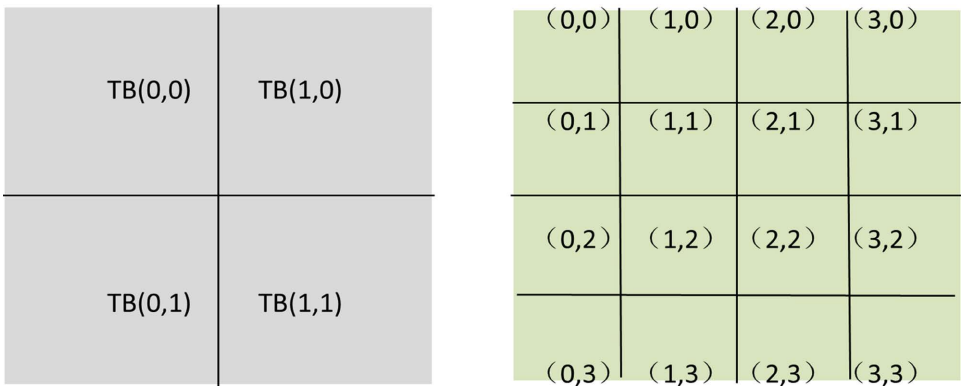

Figure 2. Thread block and thread partitioning
图 2. 线程块和线程划分

5.2. 线程块和网格配置

在进行 LU 分解时，矩阵的每一行和每一列需要进行交替的计算。我们根据矩阵规模的大小和 GPU 硬件参数来动态调整线程块尺寸，以此来利用 GPU 的并行性和计算资源利用率。线程块内的每个线程可以通过共享内存和线程同步机制来工作，而网格中的线程块则独立工作，共同完成任务。对于一个 $n \times n$ 的矩阵，可以使用一个 $n/16 \times n/16$ 的线程块配置，这样每个线程块包含 $16 \times 16 = 256$ 个线程，适合大多数中等规模大小的矩阵。如果矩阵规模较大($n \geq 8192$ 时)，系统自动切换至 $n/32 \times n/32$ 粗粒度的分块方式，

单个线程块包含了 1024 个线程，高效利用 GPU 的计算能力，如果矩阵规模较小($n \le 512$ 时)，系统则选用 8×8 的精细分块策略，通过共享内存实现数据共享，减少全局内存访问频次，即随着矩阵规模大小的变化，线程块的大小可以进行相应的调整来确保高效的利用计算资源，提高计算效率。

5.3. 通信优化策略

在 CUDA 并行 LU 分解中，通信开销主要来源于 GPU 全局内存访问和 CPU 与 GPU 之间的数据传输。为了降低通信开销，本文采取了以下优化措施：由于全局内存访问延迟较高，通常需要数百个时钟周期才能完成一次读取或写入，而共享内存(Shared Memory)访问延迟仅需几十个时钟周期。因此，直接从全局内存读取数据进行计算会导致大量的延迟，降低并行计算的效率。在 LU 分解过程中，每一轮迭代需要用到当前主元行(PivotRow)和主元列(PivotColumn)。如果所有线程直接从全局内存读取这些数据，访问延迟较高。因此，我们可以使用共享内存缓存主元行和主元列，让同一个线程块内的所有线程共享这些数据，从而减少全局内存的访问次数。采用一个线程块中的部分线程负责将主元行、主元列加载到共享内存，__shared__ float pivotRow [N]；__shared__ float pivotCol [N]；加载完成后同步(__syncthreads())，然后其他线程直接从共享内存读取数据进行计算，从而避免多次重复访问全局内存。这样，每个线程块内部的线程可以直接从共享内存中获取数据，而不是反复访问全局内存，从而大幅降低通信开销。

6. 实验结果与分析

6.1. 实验平台

为了验证基于 CUDA 的并行 LU 分解算法的有效性，本文在实验环境 GPU：NVIDIA GeForce RTX 4060，显存：8 GB GDDR6，计算能力：8.0，最高处理频率：2.4 Ghz，CPU：AMD Ryzen9 7945HX，主频为 2.5 Ghz，内存：64 GB，系统环境：Windows 10，IDE：Visual Studio 2019，CUDA Toolkit：CUDA 11.2 上进行了实验，对比了并行算法与 CPU 单线程串行算法的性能差异，并分析了不同矩阵规模大小对加速效果的影响。

6.2. 实验结果与分析

加速比是衡量并行计算效率的核心指标，它的定义为：$S = \frac{T_{\text{serial}}}{T_{parallel}}$，其中 T_{serial} 是串行算法的执行时间，$T_{parallel}$ 是并行算法的执行时间包括了通信时间。实验选取了随机稠密矩阵作为测试数据，矩阵规模从 64×64 到 2048×2048，统计了 LU 分解求解线性方程组的计算时间，单位为毫秒以及计算加速比。实验中的串行 LU 求解耗时与并行 LU 求解耗时所得数据均为随机稠密矩阵规模大小不变的情况下，随机运行 10 次所得数据的平均值。实验数据对比如表 1、表 2 所示。

Table 1. Serial vs. parallel execution time comparison (CUDA)
表 1. 串行与并行的求解时间对比(CUDA)

系数矩阵大小	串行 LU 求解耗时(ms)	CUDA 并行(ms)	加速比
64 × 64	1.37	2.40	0.57
128 × 128	11.03	8.23	1.34
256 × 256	87.04	30.35	2.86
512 × 512	700.55	116.79	5.99
1024 × 1024	5509.17	424.53	12.97
2048 × 2048	45376.68	1335.51	33.97

Table 2. Serial vs. parallel execution time comparison (OpenMP)
表 2. 串行与并行的求解时间对比(OpenMP)

系数矩阵大小	串行 LU 求解耗时(ms)	OpenMP 并行 (ms)	加速比
64 × 64	1.37	1.82	0.75
128 × 128	11.03	9.34	1.18
256 × 256	87.04	45.53	1.91
512 × 512	700.55	194.46	3.60
1024 × 1024	5509.17	1202.15	4.58
2048 × 2048	45376.68	8534.28	5.31

表 1 展示了在 CUDA 中，表 2 展示了在 OpenMP 中，不同规模大小下矩阵串行算法(CPU)与并行算法(GPU)的运行时间对比。实验结果表明，随着矩阵规模的增大，CUDA 并行算法的加速效果明显优于 OpenMP 并行算法。例如，对于 2048×2048 的矩阵，串行算法耗时约 45.3 秒，CUDA 并行算法仅需 1.3 秒，加速比约达到 34 倍，但是 OpenMP 并行算法需要 7.5 秒；对于较小规模的矩阵如 256×256，CUDA 并行算法加速比相对较低，约为 3 倍，但仍优于 OpenMP 并行算法。而当矩阵规模为 64×64 时，反而串行求解时间比 CUDA 并行求解时间短，此时加速比小于 1，仅达到 0.57，造成这个结果可能是线程同步、通信、负载均衡以及内存访问模式尚未达到标准，导致并行效率差，求解速度不如传统 CPU 计算。

7. 结论

本文通过设计基于 CUDA 的并行化 LU 分解算法，提出迭代并行与动态分块策略，优化 GPU 内存访问，实现 LU 分解的混合并行分解，在 128×128 规模以上的矩阵中，打破了传统串行算法的计算性能瓶颈。实验表明，在单 GPU 环境下，2048×2048 的矩阵求解加速比达 34 倍，验证了算法在大规模问题中的高效求解能力，成功解决了大规模稠密线性方程组求解效率低下的问题。

参考文献

[1] 阳王东, 王昊天, 张宇峰, 等. 异构混合并行计算综述[J]. 计算机科学, 2020, 47(8): 5-16.

[2] 陈国良, 孙广中, 徐云, 等. 并行计算的一体化研究现状与发展趋势[J]. 科学通报, 2009, 54(8): 1043-1049.

[3] 汪茂, 谭捍东, 林昌洪, 等. 基于大地电磁二维反演的 MPI 和 CUDA 并行算法研究[J]. 科学技术与工程, 2017, 17(10): 225-230.

[4] 张健, 李瑞田, 邓亮, 等. 面向多核 CPU/众核 GPU 架构的非结构 CFD 共享内存并行计算技术[J]. 航空学报, 2024, 45(7): 113-126.

[5] 杨梅, 李志民, 曹大勇. CUDA 架构下大规模稠密线性方程组的并行求解[J]. 计算机工程与应用, 2011, 47(32): 27-30.

[6] Lioen, W.M. and Winter, D.T. (1992) Solving Large Dense Systems of Linear Equations on Systems with Virtual Memory and with Cache. *Applied Numerical Mathematics*, **10**, 73-85. https://doi.org/10.1016/0168-9274(92)90056-j

[7] Dumas, J., Gautier, T., Pernet, C., Roch, J. and Sultan, Z. (2016) Recursion Based Parallelization of Exact Dense Linear Algebra Routines for Gaussian Elimination. *Parallel Computing*, **57**, 235-249. https://doi.org/10.1016/j.parco.2015.10.003

[8] 徐鹤, 周涛, 李鹏, 等. 基于鲲鹏处理器的 LU 并行分解优化算法[J]. 计算机科学, 2024, 51(9): 51-58.

[9] 庞文豪, 王嘉伦, 翁楚良. GPGPU 和 CUDA 统一内存研究现状综述[J]. 计算机工程, 2024, 50(12): 1-15.