南開大學

GPU 编程

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 项目仓库位置

完整代码见 github 仓库（点击跳转）

2 问题概述

2.1 高斯消去算法概述

高斯消去法（Gaussian Elimination）是一种用于求解线性方程组、计算矩阵秩以及求逆矩阵的常用方法。其主要思想是通过初等行变换将增广矩阵化为行阶梯形矩阵，从而简化求解过程。

2.1.1 步骤说明

1. **消元阶段（Forward Elimination）**：
 通过初等行变换，将矩阵化为上三角形式。此过程的目标是消去主对角线以下的元素。
2. **回代阶段（Back Substitution）**：
 从最后一行开始，逐个求解变量值。

2.1.2 初等行变换

行交换：交换两行；行倍加：将一行的若干倍加到另一行；行缩放：将某一行乘以一个非零常数。

2.2 串行算法原理分析

串行算法通过消去过程将系数矩阵转化为上三角形式，利用消元因子逐步消除下方变量，确保每行仅保留当前主元及右侧变量。回代过程利用上三角特性，从末行开始逐层代入已知变量，最终求出所有解。此方法时间复杂度为 $O(n^3)$，其中消去过程时间复杂度为 $O(n^3)$，主要进行行消元，有三重循环，回代过程时间复杂度为 $O(n^2)$，主要进行逆向求和，有两重循环。

```
for (int k = 0; k < n; k++) {
 for (int i = k + 1; i < n; i++) {
 double factor = A[ i ][k] / A[k][k];
 for (int j = k + 1; j < n; ++j) {
 A[ i ][ j ] -= factor * A[k][ j ]; }
```

```
 b[ i ] -= factor * b[k];}}
x[n - 1] = b[n - 1] / A[n - 1][n - 1];
 for (int i = n - 2; i >= 0; i--) {
 double sum = b[ i ];
 for (int j = i + 1; j < n; j++) {
 sum -= A[ i ][ j ] * x[j ];}
 x[ i ] = sum / A[ i ][ i ];}
```

3 实验介绍

高斯消元作为一种经典的线性方程组求解方法，其核心包括主元选取、逐行消元和回代求解等步骤。该算法计算复杂度为 $O(n^3)$，在处理大规模稠密矩阵时常成为性能瓶颈。为提升其计算效率，本实验采用基于 CUDA 的 GPU 并行编程方法，对高斯消元算法进行加速优化。

本实验的平台是北京超算平台，GPU 型号是 NVIDIA T4，环境是 GT-Ubuntu22.04-CMD-V3.0。

本实验的目标是完成高斯消元的 CUDA 加速实现，并对任务分配策略、线程块数量和线程块大小等参数进行实验设计和性能评估。通过分析不同并行粒度下的执行效率，探索优化内存访问模式与线程调度策略对整体性能的影响。

CUDA 编程模型的主要特点包括：大规模并行性；层次化线程结构；多级内存层次；高吞吐量计算：适用于矩阵运算、图像处理、物理仿真等高性能计算场景。

4 GPU 设计实现

首先，在主机端初始化系数矩阵 A 与常数列向量 b，采用一维数组表示二维矩阵结构，以便在 GPU 中进行线性化访问。随后使用 CUDA 的 cudaMalloc 与 cudaMemcpy 接口将数据拷贝到设备端显存中。

核心计算部分采用了每个线程块 Block 负责矩阵中的一行，线程 Thread 负责该行中的不同列元素更新的任务分配方式。具体而言，核函数 division_kernel_row_col 中通过 blockIdx.x 确定当前 Block 对应的消元行号，通过 threadIdx.x 决定本线程更新的列元素。在核函数中，每个线程根据高

斯消元的公式，对矩阵元素 A[i][j] 执行如下操作：

$$A_{ij} = A_{ij} - \frac{A_{ik}}{A_{kk}} * A_{kj}$$

该更新逻辑通过并行的列级线程并发执行，在共享内存冲突较少的情形下可以有效提升吞吐量。

为了避免线程间的数据写冲突与冗余计算，线程块内部在更新常数列 b 与主元列 A[i][k] 时，仅由 threadIdx.x == 0 的线程负责执行，从而实现对消元行的原子级更新。

在主程序中，使用 cudaEventRecord 实现 GPU 代码段的运行时间测量，确保性能评估的准确性。所有消元步完成后，将结果从设备端拷贝回主机端，并使用 CPU 完成反向替代（回代）过程以得到最终解向量 x。

5 对比分析方向

通过不同问题规模（2 的整数次方）下的程序执行时间以及加速比进行不同优化策略的对比分析：

1. 串行算法与 GPU 并行算法对比分析
2. 不同任务划分方式对比分析
3. 不同线程块数/线程块大小对性能的影响
4. 其他探索与分析

6 实验结果分析

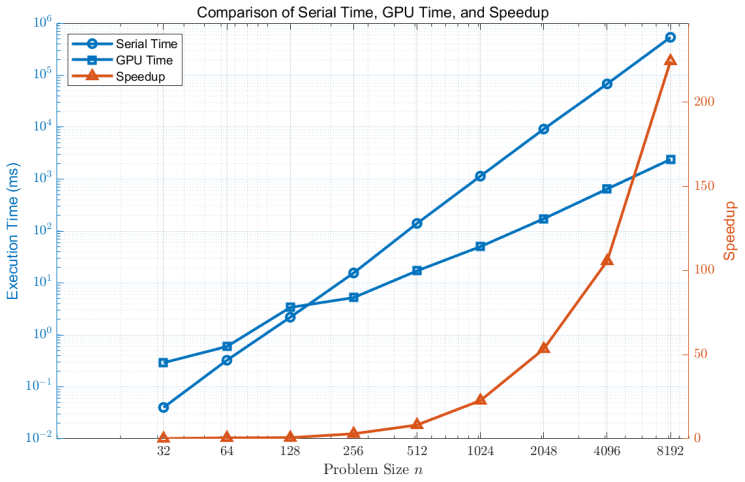
6.1 串行算法与 GPU 并行算法对比分析

在不同问题规模下, 对串行与 GPU（blocksize=256）高斯消元执行时间与加速比进行分析。

这里的 GPU 高斯消元算法采用的是策略 B，具体在“不同任务划分方式对比分析”部分进行介绍。

表 1: 串行时间与 GPU 时间对比及加速比

问题规模 n	串行时间 (ms)	GPU 时间 (ms)	加速比
32	0.040	0.291072	0.1374
64	0.325	0.597056	0.5443
128	2.188	3.3792	0.6475
256	15.568	5.2239	2.9800
512	140.059	17.1272	8.1776
1024	1139.062	50.1558	22.7100
2048	9196.672	172.323	53.3688
4096	68115.675	644.931	105.617
8192	537561.435	2394.45	224.500



小规模下 GPU 开销难摊薄，加速比 <1；n 256 后并行优势显现，n=8192 时加速比达 224.5×。

上述现象的本质原因在于，高斯消元算法中的行操作在消元过程中可以高度并行化，而 GPU 正适合这样的结构化数据并行任务。因此，综合分析可以得到以下结论：

1. GPU 并行算法在大规模计算问题中具有显著优势，能提供数量级的性

能提升。

2. 对于小规模问题，串行算法更具优势，因为 GPU 的调度与内存开销在此时难以抵消。

3. 加速比的提升依赖于问题规模和算法的并行度，高效的线程调度、合适的线程块设计与内存访问策略是优化 GPU 程序性能的关键。

6.2 不同任务划分方式对比分析

在基于 CUDA 的高斯消元算法中，任务分配策略直接影响程序的并行效率和资源利用率。本实验主要实现并比较了两种典型的任务分配方式：

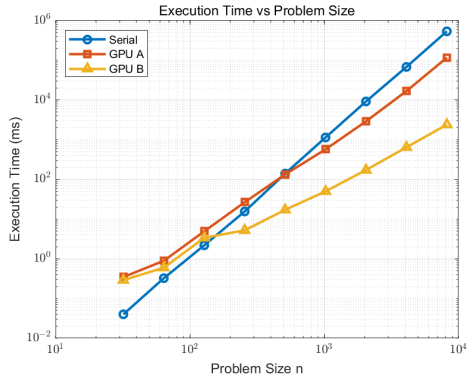
- 策略 A 将每线程映射到一行消元，线程独立执行，但缺乏协调，易造成资源浪费，难以充分利用 GPU 并行性。
- 策略 B 将一行分配给一个 block，线程处理不同列，细粒度并行提升效率，适合大矩阵，资源利用更充分。

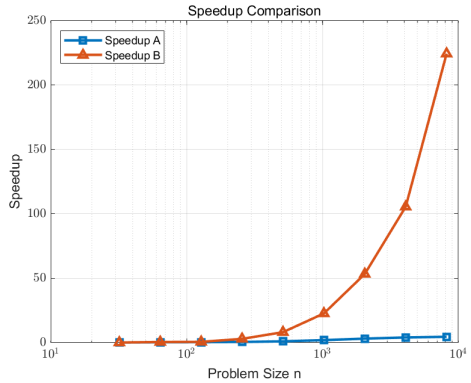
下面是策略 A 和策略 B 在线程块数量为 256 时的实验数据:

表 2: 串行与策略 A 和 B 的执行时间与加速比对比

问题规模	串行时间 (ms)	A 时间 (ms)	A 加速比	B 时间 (ms)	B 加速比
32	0.04	0.347328	0.1152	0.291072	0.1374
64	0.325	0.892032	0.3645	0.597056	0.5443
128	2.188	4.92394	0.4445	3.3792	0.6475
256	15.568	26.6504	0.5843	5.2239	2.9800
512	140.059	133.426	1.0497	17.1272	8.1776
1024	1139.062	571.8	1.9917	50.1558	22.710
2048	9196.672	2902.44	3.1683	172.323	53.3688
4096	68115.675	16773	4.0636	644.931	105.617
8192	537561.435	116790	4.6016	2394.45	224.5

加速比以及加速比对比折线图如下：





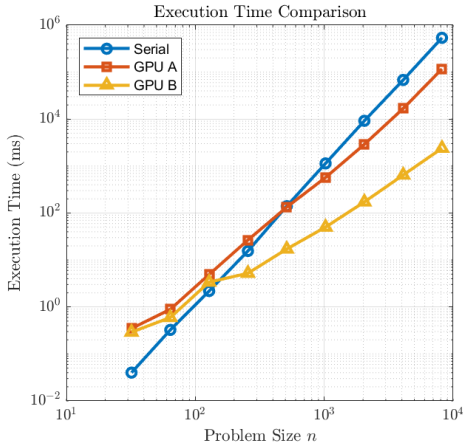
- 小规模下 GPU 初始开销大，加速比低于 1，反而慢于串行 CPU。
- n 256 后 GPU 优势显现，策略 B 在 n=8192 时加速比远超策略 A。
- 策略 A 线程处理整行，数量不足且缺乏并发，资源利用率低。
- 策略 B 线程处理列，提升并发度至 $O(n^2)$，共享内存减少访存。
- B 线程读数据连续，访存对齐好，A 线程读写不连续，访存效率低。
- B 更好利用 SIMD 并行，操作统一高效；A 无协作，指令复杂低效。
- B 适配性强，不依赖维度配置；A 耦合严重，扩展需手动调整。

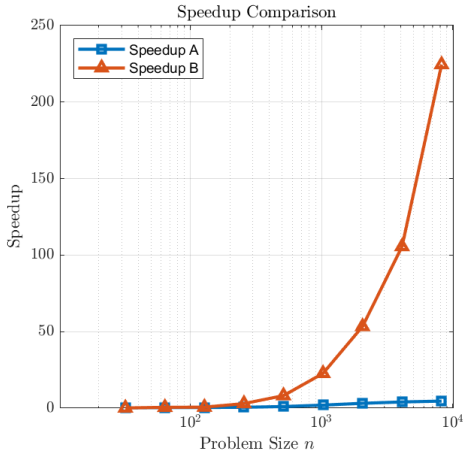
6.3 不同线程块数/线程块大小对性能的影响

blocksize 是 CUDA 中控制线程调度与资源分配的关键参数。为评估其对性能的影响，我在策略 B 下测试了 blocksize 为 64 至 1024 的配置，保持线程总数覆盖问题规模，通过调整 blockDim.x 与 gridDim.x 协同完成任务，分析粒度划分对执行效率的影响。

表 3: 串行与策略 A 和 B 的执行时间与加速比对比

问题规模	串行时间 (ms)	A 时间 (ms)	A 加速比	B 时间 (ms)	B 加速比
32	0.04	0.347328	0.1152	0.291072	0.1374
64	0.325	0.892032	0.3645	0.597056	0.5443
128	2.188	4.92394	0.4445	3.3792	0.6475
256	15.568	26.6504	0.5843	5.2239	2.9800
512	140.059	133.426	1.0497	17.1272	8.1776
1024	1139.062	571.8	1.9917	50.1558	22.710
2048	9196.672	2902.44	3.1683	172.323	53.3688
4096	68115.675	16773	4.0636	644.931	105.617
8192	537561.435	116790	4.6016	2394.45	224.5





- 小规模问题启动代价大，各 blocksize 加速比 <1，慢于串行程序。
- 中等规模起 blocksize 差异显现，64 与 128 加速提升明显，并行快。
- blocksize=64 最优，SM 利用充分，线程调度灵活，加速比 >660×。
- blocksize 过大导致 block 数少、共享资源紧张，性能反降。
- blocksize=64/128 稳定性强，64 在多数情况下表现最佳。

6.4 其他探索

在完成策略 B 的基础上，我进一步尝试了两种不同的优化方向，希望在保持并行度的同时进一步压缩内存访问开销与同步成本。然而，实验发现这些优化策略的性能表现反而不如未优化前的策略 B，现分析如下：

6.4.1 策略一：共享内存与寄存器重构

本实验引入寄存器缓存、共享内存与编译器提示等优化手段，理论上可提升访存效率。但实际中因频繁同步开销大、共享内存压力增大及优化收益不足以抵消资源竞争，整体性能提升有限。

6.4.2 策略二：更实用的策略——每线程处理整行

该策略将每个线程负责一整行的更新工作，意图规避共享内存与线程同步问题：每个线程独立处理整行，完全避免线程间同步；内存访问按行进

行，具有良好的 coalesced 访问模式；不依赖共享内存，避免 bankconflict 和容量限制。

然而该策略在大规模矩阵上的性能表现仍低于策略 B，原因包括：线程分布不均、线程数量有限、指令压力大。

7 总结反思

本实验实现了高斯消元的 CUDA 并行化，比较串行与 GPU 策略，深入理解任务划分、线程调度与内存优化对性能的影响。实验表明 GPU 在大规模计算中具有明显优势，策略 B 通过细粒度划分显著优于策略 A。blocksize=64 表现最优，资源利用充分。优化过程中发现性能提升依赖实现细节，盲目优化可能引发瓶颈。整体而言，本实验加深了我对 CUDA 模型与 GPU 架构的理解，认识到并行优化需算法与硬件协同，未来可探索稀疏矩阵与动态调度等进一步提升性能。