

南开大学

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 | 3 |
| 2 | 问题概述 | 3 |
| 2.1 | 高斯消去算法概述 | 3 |
| 2.1.1 | 步骤说明 | 3 |
| 2.1.2 | 初等行变换 | 3 |
| 2.1.3 | 示例 | 3 |
| 2.2 | 串行算法原理分析 | 4 |
| 3 | 实验介绍 | 5 |
| 4 | GPU 设计实现 | 5 |
| 5 | 对比分析方向 | 6 |
| 6 | 实验结果分析 | 6 |
| 6.1 | 串行算法与 GPU 并行算法对比分析 | 6 |
| 6.2 | 不同任务划分方式对比分析 | 8 |
| 6.3 | 不同线程块数/线程块大小对性能的影响 | 11 |
| 6.4 | 其他探索 | 14 |
| 6.4.1 | 策略一：共享内存与寄存器重构 | 14 |
| 6.4.2 | 策略二：更实用的策略——每线程处理整行 | 15 |
| 7 | 总结反思 | 15 |

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

2.1.3 示例

设线性方程组为：

$$\begin{cases} 2x + y - z = 8 \\ -3x - y + 2z = -11 \\ -2x + y + 2z = -3 \end{cases}$$

其增广矩阵为：

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array} \right]$$

通过高斯消去操作，最终可求得唯一解：

$$x = 2, \quad y = 3, \quad z = -1$$

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。

```
1 // 消去过程
2 for (int k = 0; k < n; k++) {
3     for (int i = k + 1; i < n; i++) {
4         double factor = A[ i ][k] / A[k][k];
5         for (int j = k + 1; j < n; ++j) {
6             A[ i ][ j ] -= factor * A[k][ j ];
7         }
8         b[ i ] -= factor * b[k];
9     }
10 }
11 // 回代过程
12 x[n - 1] = b[n - 1] / A[n - 1][n - 1];
13 for (int i = n - 2; i >= 0; i--) {
14     double sum = b[ i ];
15     for (int j = i + 1; j < n; j++) {
16         sum -= A[ i ][ j ] * x[j ];
17     }
18     x[ i ] = sum / A[ i ][ i ];
19 }
```

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。

CUDA (Compute Unified Device Architecture) 是 NVIDIA 提出的通用并行计算平台和编程模型，允许开发者使用 C/C++ 等语言编写可在 GPU 上并行执行的程序。与传统 CPU 串行执行模式不同，GPU 拥有数千个轻量级线程，可同时处理大量数据，特别适合高斯消元这类具有大量重复数值计算和可并行操作的任务。

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：支持成千上万线程并发运行，显著加速数据密集型任务；
- 层次化线程结构：采用线程块 (Block) 与线程网格 (Grid) 结构，支持灵活的任务划分；
- 多级内存层次：提供全局内存、共享内存、寄存器等多种存储资源，适合进行显存优化；
- 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

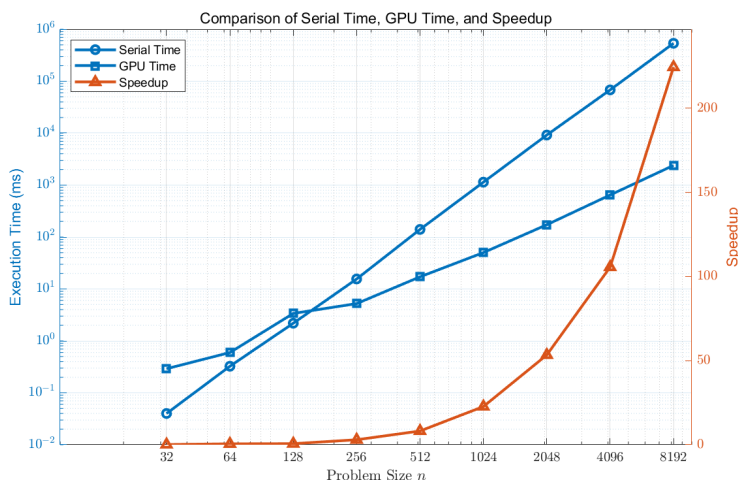
6.1 串行算法与 GPU 并行算法对比分析

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 |



首先，在问题规模较小时 ($n=32$ 128)，GPU 算法的性能明显不如串行算法。此时加速比远小于 1。这主要归因于 GPU 的内核启动开销、主机与设备之间的数据拷贝延迟、线程同步与资源调度等额外开销在小问题规模下难以被摊薄，导致总体性能低于串行算法。随着问题规模增大 ($n=256$)，GPU 算法开始展现其强大的并行计算能力。加速比逐渐超过 1，并且不断增大，当 $n=8192$ 时，甚至达到约 224.5 倍的加速比。这一趋势充分体现了

GPU 在处理大规模数据并行任务方面的高效性和可扩展性。

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。同时，GPU 的高吞吐量内存访问机制、多核 SM 单元架构以及 SIMT 执行模型也极大地加速了矩阵计算。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：

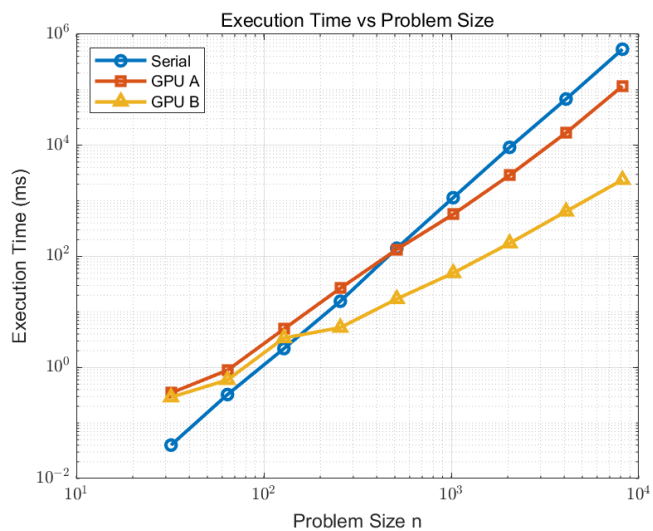
- 第一种为策略 A，即每个线程独立负责矩阵中从第 $k+1$ 行起的一整行的消元操作。该策略通过将线程一一映射到待处理的行上，利用线程独立性并行执行各行的更新。然而，该方法存在较明显的资源利用缺陷：由于线程只处理完整的行，而一个 block 内部线程之间没有进行细粒度的任务协调，部分线程可能处于空闲状态，从而导致 warp 内的执行单元浪费。此外，该策略在面对列数较多的矩阵时，无法充分利用 GPU 的并行线程能力。
- 第二种为策略 B，采用更细粒度的任务分配方式，即每个 block 负责处理一行，而该 block 内的每个线程则负责该行中不同的列元素更新。该方法通过将一行的多个元素分配给多个线程，并在 block 内部进行同步，从而更充分地发挥 GPU 的并行计算能力。由于 block 对应一行，thread 对应一列，相比策略 A，策略 B 更适用于列数较多的大规模矩阵计算，能够有效避免线程空转，提高资源利用效率。

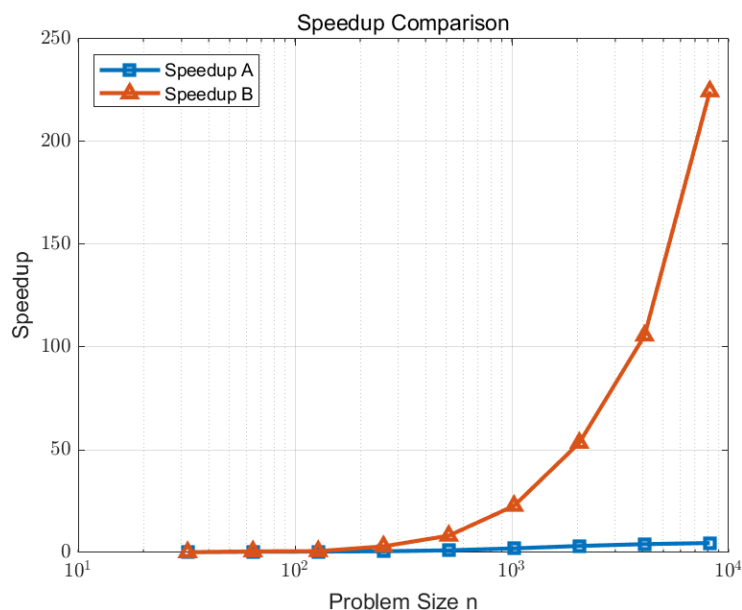
下面是策略 A 和策略 B 在线程块数量为 256 时的实验数据：

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：





- 在问题规模较小时 ($n=128$)，两种策略的加速比均明显低于 1，GPU 算法性能比串行算法要差。这种现象并不意外，GPU 并行计算存在显著的初始开销，包括核函数的调用开销、主机与设备间的数据拷贝时间、线程创建与调度延迟等。这些固定成本在小规模计算中占比很大，难以通过计算量来摊薄，从而导致整体执行时间反而高于 CPU 串行算法。
- 随着问题规模逐渐增大 ($n=256$)，GPU 的并行计算能力开始显现，两种策略的加速比都在逐步上升，特别是策略 B，加速比迅速提升。策略 B 在 $n=8192$ 时达到加速比 $224.5\times$ ，而策略 A 同期加速比仅为 $4.60\times$ 。两者性能差异在数量级上拉开了显著差距，这直接反映了任务划分策略对 GPU 性能发挥的决定性作用。
- 策略 A 将每个线程映射为一个完整的行操作，由于矩阵大小为 $n\times n$ ，线程数量最多为 n ，远低于 GPU 能承载的并行线程上限。同时，由于每行涉及 $n-k$ 个元素的逐列更新操作，在缺乏线程内并发的情况下，大量的算术运算和访存任务都压在单线程内，容易形成指令瓶颈。此外，线程之间没有任何通信与协作，warp 级资源调度形同虚设，造成部分硬件资源空闲。

- 策略 B 通过更细粒度的任务划分，提升了线程级并行度。在该策略中，每个 Block 负责一行内的消元操作，Block 内各线程并行处理该行的不同列，使得并发线程总数提升至 $O(n^2)$ 量级，更贴近 GPU 架构的最佳负载区间。与此同时，线程之间可以共享中间变量、避免冗余计算，并结合共享内存优化访存延迟，大幅度减少全局内存访问次数，提升了内存带宽利用率。
- 策略 B 对内存访问模式进行了有效对齐：由于列内线程连续读取数组元素，能够实现 memorycoalescing（内存合并访问）访问模式，大大降低访存延迟。而策略 A 中线程独立访问每行各列的数据，往往不连续，造成非对齐访存甚至 bankconflict（内存银行冲突），严重影响吞吐率。
- 此外，在执行效率方面，策略 B 能更好地利用 SIMD 结构特性，借助统一指令调度多个线程执行相似操作，在大规模矩阵操作中展现出极高的浮点运算效率。而策略 A 由于线程独立运行、无协作、操作指令复杂性更高，难以充分发挥 SIMD 架构的指令并行优势。
- 从可扩展性角度看，策略 B 的设计思路能自然适配任意矩阵大小，不依赖矩阵的具体维度对线程数进行硬编码配置，具备更强的通用性。而策略 A 的线程分布与行数高度耦合，扩展至更大问题规模时需要手动调整线程数和 Grid 划分策略，维护复杂性更高。

综上所述，策略 B 凭借其合理的线程分工与高效的内存访问模式，展现出远优于策略 A 的性能表现。特别是在大规模矩阵计算场景下，其加速比的快速增长清晰反映了并行度、指令调度、内存优化等方面协同带来的巨大性能优势。除此之外，也通过实验分析充分说明了在 GPU 编程中，任务划分策略的设计质量往往决定了算法性能的上限。

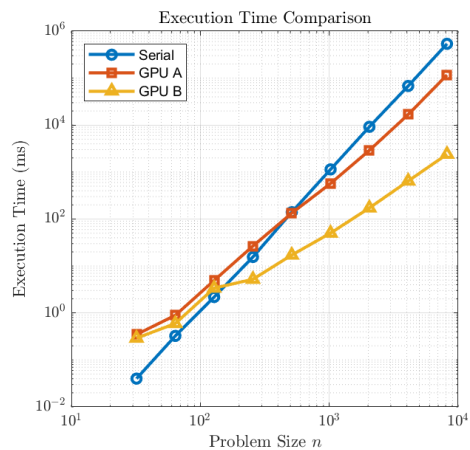
6.3 不同线程块数/线程块大小对性能的影响

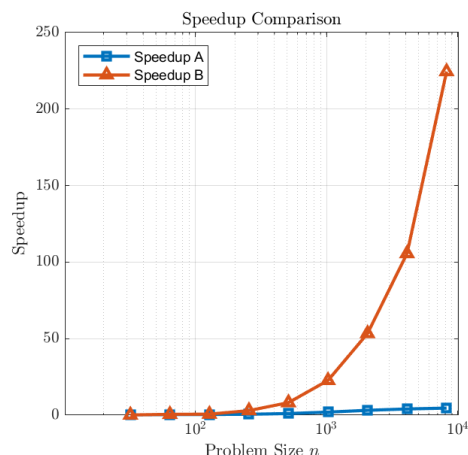
blocksize 是 CUDA 编程中控制线程调度粒度与资源分配效率的关键参数，不同的设置直接影响着线程活跃度、寄存器与共享内存分配，以及线程块在多个 StreamingMultiprocessor 上的并发调度能力。为探究不同线程块数/线程块大小配置对 GPU 性能的影响，我在保持线程总数覆盖问题规模所需的前提下，测试了策略 B 在 blocksize 为 64、128、256、512 和 1024

时的执行时间。每组实验通过调整线程块大小 (blockDim.x) 并配合合理数量的线程块 (gridDim.x) 共同完成高斯消元任务，从而评估粒度划分对并行性能的影响。

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 |





- 小规模问题下难以显现并行优势：当问题规模较小 ($n=128$) 时，所有 `blocksize` 设置的加速比均未超过 1，甚至部分情况下运行时间明显高于串行程序。与前两部分的分析相同，这一现象源于 GPU 的并行机制启动需要额外的代价。
- 中等规模开始出现分化趋势：从 $n=256$ 起，各 `blocksize` 参数间性能差异开始显现。`blocksize=64` 和 `128` 的表现稳定，并开始逐步超越串行程序，体现出并行优势，而更大的 `blocksize` 设置尚未显著提升性能。这表明合理的线程块配置能更早发挥 GPU 并行的潜力。
- `blocksize=64` 拥有最优可扩展性：当问题规模进一步增大至 $n=8192$ ，`blocksize=64` 获得了最高的加速比（超过 $660\times$ ），远超其他参数。这种表现归因于更小的 `blocksize` 能划分出更多 block，使得 GPU 的 SM 数量得以更充分利用，同时也提升了线程调度灵活性。每个 block 内线程数量少，有利于 warp 组织与共享内存合理分配，减少线程间的依赖与空转。
- `blocksize` 过大导致资源利用率下降：当 `blocksize` 增加至 512 或 1024 时，程序性能显著下降。这是因为每个 block 中线程数过多，线程块总数变少，限制了在多个 SM 上的并发调度数量。此外，大 `blocksize` 配置可能导致共享内存不足、寄存器压力升高，从而引发资源争用与频繁溢出到全局内存，进一步降低吞吐率。
- `blocksize=64` 和 `128` 的稳定性与鲁棒性更强：这两种设置在从中小到

大规模的全阶段中都表现出了较强的可扩展性和稳定的性能优势，在当前的高斯消元场景下，能够提供更好的并行度与资源利用的平衡点。其中 $\text{blocksize}=64$ 综合性能最优，并行效率最高，在部分问题规模下实现超过 $660\times$ 的加速，并且我相信随着问题规模继续增大，加速幅度还会进一步提升。

综上所述，线程块大小的设置对 GPU 程序的加速效果具有相当关键的作用。合理设置 blocksize ，不仅要考虑线程数量和并行度，还需兼顾 GPU 硬件资源（如 SM 数量、共享内存容量、寄存器数）的限制和调度机制。较小的 blocksize 更能发挥 GPU 架构的优势，提高并发调度效率，适配高斯消元这类高内存访问、结构规律明确的线性代数运算，从而获得显著的性能提升。

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

1. 优化点 1: A_{ik} 与 A_{kk} 的寄存器缓存。原始策略中每个线程都需要访问一次 A_{ik} 和 A_{kk} ，这属于全局内存的重复访问，效率较低。通过将些值缓存到寄存器中，并通过线程内变量进行共享，可避免重复访问，提高访问速度。
2. 优化点 2: A_{kj} 缓存到共享内存。第 k 行的各列数据会被多个线程访问，通过将其缓存到共享内存，可大幅减少对全局内存的带宽需求。
3. 优化点 3: 使用 `__restrict__` 和 `const` 关键字。这类语义提示可帮助编译器更积极地进行加载优化与指令重排，在理论上能提高运行效率。

尽管上述优化理论上能减少冗余访存、加速访问路径，但实验中发现：

1. 同步开销上升：由于共享内存的数据由多个线程协同加载并使用，且每一轮消元都需要 `barrier` 同步，频繁同步操作抵消了访存带宽优化带来的收益，尤其在列数较大（宽矩阵）的情况下更明显。

2. 共享内存压力增加：在问题规模增大时，第 k 行需要缓存的元素数量成倍增长，可能导致共享内存容量受限，甚至引发 bankconflict 或 fallback 到慢速全局内存。
3. 优化收益不足以抵消代价：线程间合作带来的调度和寄存器竞争，以及 GPU 编译器在复杂指令路径下可能无法完全完成访存重排，使得实际收益远低于预期。

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：

1. 每个线程独立处理整行，完全避免线程间同步。
2. 内存访问按行进行，具有良好的 coalesced 访问模式。
3. 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

综上所述，两种优化策略虽然理论上能减少冗余访存与同步，但实际上引入了线程协同与资源竞争等新的瓶颈，无法达到策略 B 中高并发、负载均衡的优势。

7 总结反思

本实验围绕高斯消元算法的 CUDA 并行化实现展开，通过对串行版本与多种 GPU 策略的对比分析，深入理解了并行编程中的任务划分、线程调度与内存优化等核心要素对程序性能的深刻影响。

首先，从串行算法到 GPU 并行算法的迁移过程中，尽管 GPU 在初期小规模问题上存在较高的调度和通信开销，导致整体性能不如串行版本，但随着问题规模的扩大，GPU 展现出强大的并行计算能力和显著的加速效果，

尤其在 $n = 8192$ 时实现超过 $200\times$ 的加速比。这验证了 GPU 对于大规模、结构规律性强的数值计算任务具有天然优势。

其次，通过对比策略 A 与策略 B，可以明显看出任务划分的粒度对性能的决定性作用。策略 B 采用了更细粒度的任务分配方式，有效提升了线程活跃度和资源利用率，在内存访问模式、线程协同和负载均衡方面均优于策略 A，是更适合 GPU 架构的实现方式。实验数据也充分表明，优秀的任务划分策略能够将 GPU 的潜力发挥到极致。

在线程块大小的选择上，实验发现较小的 blocksize（如 64）具有更好的可扩展性和更高的并发度，能更充分利用 GPU 的多核资源和调度机制，从而获得更优的性能表现。与此同时，过大的 blocksize 反而会引发资源竞争、线程调度不灵活等问题，导致性能下降。

最后，尽管尝试了一些进一步的优化（如共享内存与寄存器重构、避免线程间同步等），但实际结果表明优化策略的效果高度依赖于实现细节与资源配置。不合理的优化反而可能引入新的瓶颈，如同步开销增加、共享内存溢出等，提醒我们并行程序优化应慎重权衡成本与收益，避免过度设计。

总体而言，本实验不仅完成了高斯消元算法的 GPU 并行加速实现，还通过丰富的对比实验和详实的数据分析，深入理解了 CUDA 编程模型中影响性能的关键因素，如线程划分策略、内存访问模式、block 配置与同步机制等。通过实践，我对 GPU 的体系结构与编程模式有了更加直观而深入的认识，也意识到性能优化是一项系统性工程，需从算法层面与硬件资源匹配的角度共同设计。未来若能结合稀疏矩阵优化、动态线程调度等技术，相信在更广泛的线性代数场景中仍有巨大性能提升空间。