南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计

多线程编程 (Pthread&OpenMP)——高斯消去算法

孟启轩 2212452

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2025 年 5 月 28 日

摘要

结合课堂内容以及实验指导书，我采用 Pthread 和 OpenMP 对高斯消元算法并行优化，在 x86 平台和 ARM 平台实现了不同的优化策略，采用不同任务划分方式，还与 SIMD 进行结合，并测试执行时间计算加速比来对比分析性能。

关键字：多线程 Pthread OpenMP 高斯消元 ARM x86 SIMD 并行

目录

一、 问题概述

(一) 高斯消去算法概述

高斯消去算法是求解线性方程组 Ax = b 的经典算法，其核心分为两个阶段，消去过程和回代过程。

消去过程：将系数矩阵 A 转换为上三角矩阵

1. **归一化**：对第 k 行进行除法操作，使主元 $A[k,k]$ 变为 1。

$$A[k,j] \leftarrow \frac{A[k,j]}{A[k,k]} \quad \text{对所有列} j \geq k.$$

2. **消去**：利用第 k 行对后续行（$k+1$ 至 N 行）执行减法操作，消除这些行在第 k 列的元素。

$$A[i,j] \leftarrow A[i,j] - A[i,k] \cdot A[k,j] \quad \text{对所有行} i > k \text{ 和列} j \geq k.$$

回代过程：求解未知数 x_i

1. 从最后一行开始，逆向逐行求解未知数 x_i。

$$x_i \leftarrow \frac{b_i - \sum_{j=i+1}^{n} A[i,j] \cdot x_j}{A[i,i]} \quad \text{对} i = N, N-1, \ldots, 1.$$

(二) 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。

串行算法核心代码

```
// 消去过程
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];
    }
}
// 回代过程
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];
}
```

二、 实验环境

实验环境分别为华为鲲鹏服务器以及笔记本电脑，主要硬件信息如下：

属性	**Kunpeng-920（ARM 服务器）**	**AMD Ryzen 7 5800H**
架构	ARMv8 (aarch64)	x86_64
指令位宽	64-bit	64-bit
核心数	8	8
线程数	8（1 线程/核）	16（2 线程/核，支持 SMT）
主频（Base）	2.6GHz，BogoMIPS: 200	3.2 GHz
一级缓存 / L1 Cache	128 KB	64 KB
二级缓存 / L2 Cache	512 KB	512 KB
三级缓存 / L3 Cache	48 MB	16 MB
厂商	HiSilicon（华为）	AMD
型号	Kunpeng-920	Ryzen 7 5800H
缓存/内存带宽支持	高并发设计，专为服务器优化	高速缓存优化，适合高性能计算
支持指令集扩展	NEON, SHA, AES, CRC 等	SSE, AVX2, FMA 等
用途定位	服务器/数据中心	笔记本/移动平台

三、 实验介绍

高斯消元作为一种经典的线性方程组求解方法，其核心包括标准化主元行、逐行消元与回代等步骤，计算量大、数据访问密集，极易成为程序的性能瓶颈。为提升其运行效率，实验将利用 **Pthread 和 OpenMP** 两种主流并行编程模型对高斯消元算法进行加速优化：

- **Pthread** 提供了较底层的线程创建与管理接口，具备更高的灵活性与精细控制能力，适用于自定义线程行为与任务划分；
- **OpenMP** 作为一种基于编译指令的高层抽象并行框架，能够方便地在现有串行程序中插入并行结构，简洁易用，适合快速实现数据并行和任务并行。

矩阵初始化设计

为了避免在高斯消元过程中出现 `inf` 或 `nan` 的数值问题，本实验在矩阵初始化阶段进行了特别设计。传统的随机初始化方式可能会在矩阵规模较大时生成病态或非满秩矩阵，进而引发数值不稳定，影响算法的正确性与并行性能。为此，本实验采用了两阶段的初始化策略：

- **阶段一：构造上三角矩阵**。首先将矩阵 A 初始化为上三角形式，即仅填充对角线及其上方元素，确保矩阵具有良好的数值特性，避免除以过小数值造成精度丢失。对角线元素初始化为 `1000+rand%100` 的较大值，用以增强主元稳定性。
- **阶段二：行线性组合扰动**。随后，程序从矩阵中随机抽取若干行，进行线性组合扰动操作，即执行若干轮 $A_{row1} \leftarrow A_{row1} + \alpha \cdot A_{row2}$，其中 α 为 $[0, 1)$ 区间的随机系数。这一步骤增强了矩阵的复杂性与通用性，同时保持其可解性，避免生成非满秩矩阵。

向量 b 的初始化方式与矩阵 A 保持一致，在每次行变换后也同步进行线性组合，确保方程组结构保持一致，有效保障了后续串行及并行高斯消元过程中数值稳定性和实验结果的有效性。

矩阵初始化代码

```
void initialize_matrix() {
    srand(time(0));
    for (int i = 0; i < n; ++i) {
        fill(A[i], A[i] + n, 0.0);
        A[i][i] = rand() % 100 + 1000;
        for (int j = i + 1; j < n; ++j) {
            A[i][j] = rand() % 100 + 1;
        }
        b[i] = rand() % 100 + 1;
    }
    for (int k = 0; k < n / 2; ++k) {
        int row1 = rand() % n;
        int row2 = rand() % n;
        double factor = (rand() % 100) / 100.0;
        for (int j = 0; j < n; ++j) {
            A[row1][j] += factor * A[row2][j];
        }
        b[row1] += factor * b[row2];
    }
}
```

四、 Pthread 设计实现

根据实验指导书主要设计以下四种 Pthread 优化策略。

(一) 静态线程 + 信号量同步

该策略在程序启动时一次性创建多个工作线程（静态线程池），主线程在每轮消元中执行除法操作，之后通过信号量 `sem_post()` 唤醒所有工作线程。各个工作线程在被唤醒后，执行自己负责的消去任务，完成后通过 `sem_post()` 通知主线程自己已经完成。主线程在收到所有线程的完成信号后，进入下一轮迭代。此策略通过主线程控制节奏，采用信号量机制实现线程间同步，避免频繁创建销毁线程，可以提高整体效率。

(二) 静态线程 + 信号量同步 + 三重循环全部纳入线程函数

该策略在（一）的基础上进一步将整个三重循环结构都纳入工作线程内部。在每轮迭代中，线程 `t_id = 0` 执行除法操作，其它线程通过信号量 `sem_wait()` 等待除法完成后开始消去，并在消去完成后通过信号量与主线程和其他线程协同进入下一轮。此策略减少了主线程的逻辑复杂度，强化了线程的自治性，整个迭代流程中主线程不直接参与计算，仅起到同步协调作用。

(三) 静态线程 + barrier 同步

该策略同样使用静态线程池，不使用信号量，而使用 `pthread_barrier_t` 实现多线程间的阶段同步。在每轮迭代中，线程 0 完成除法操作后，所有线程在 `barrier_Division` 处同步，统一进入消去阶段。消去阶段采用行划分或列划分方式实现并行，完成后在 `barrier_Elimination` 处再次同步进入下一轮。此策略结构清晰、同步简单，适合多阶段协同并行计算。

(四) 动态线程

该策略不使用线程池，而是在每一轮消元迭代中动态创建若干线程，每个线程负责一行的消去操作。线程在完成任务后立即退出，主线程通过 `pthread_join()` 等待所有线程结束，然后进入下一轮。此策略实现逻辑简单，适合线程生命周期短、任务划分明显的场景，但每轮都存在线程创建与销毁开销，适用于线程管理资源充足、线程数量较少时的情况。

五、 OpenMP 设计实现

OpenMP 是一种基于共享内存的并行编程模型，采用编译指令方式进行并行化，使用成本低，适合快速构建和测试并行程序。在本实验中，OpenMP 被用于加速高斯消元中的主元标准化与矩阵消元过程。

(一) 并行结构设计

OpenMP 的并行优化主要集中在高斯消元的两层循环中。最外层迭代变量 k 表示当前的主元行，每一轮迭代中包括两个主要操作：

- **标准化主元行**：由单个线程串行完成（使用 `#pragma omp single`），避免写冲突；
- **并行消元**：对主元行以下的若干行进行矩阵更新操作，可并行执行。

程序整体结构通过 `#pragma omp parallel` 仅创建一次线程池，避免反复创建销毁线程所带来的开销。每轮迭代中，利用 `omp for` 对任务进行分发，保证并行部分充分调度多个线程运行。

(二) 同步机制与数据一致性

OpenMP 中线程间共享数据结构 A 和 b，而每轮迭代都需在标准化主元行之后再进行消去操作。为了保证正确的执行顺序，使用 `single` 语句块和隐式 barrier（或显示的 `#pragma omp barrier`）保证同步，使得每轮计算阶段清晰划分为串行标准化与并行消去两部分，确保数据一致性。

(三) 回代阶段处理

考虑到回代阶段中，当前行计算依赖于下一行的结果，因此本实验将回代部分保持为串行执行。虽然存在一定的并行化可能性，但由于该阶段计算量远小于前向消元阶段，影响较小，且保留串行有助于简化同步和验证。

(四) 调度策略与 chunk_size 设置

在实验中，为进一步探索 OpenMP 的性能调度机制，设置了不同的 `schedule` 策略，`static`、`dynamic` 和 `guided`，并对比了不同 `chunk_size` 设置下程序的运行时间表现。

`static` 适合任务粒度平均、负载均衡性高的情况，而 `dynamic` 和 `guided` 更适合处理不均匀任务，如列划分或后期矩阵稀疏变化剧烈的情况。实验通过设置 `chunk_size` 为 $1, 4, 8, \ldots, 256$ 等多个级别进行对比测试，以评估调度粒度对性能的影响。

六、 对比分析方向

通过不同问题规模（2 的整数次方）下的程序执行时间进行不同优化策略的对比分析。

(一) Pthread

x86 平台

- 三种静态线程优化策略对比
- 静态优化策略不同线程数对比
- 静态线程与动态线程对比
- 不同任务划分策略（行/列划分）对比

ARM 平台

- 三种静态线程优化策略对比
- 静态优化策略不同线程数对比
- 结合 SIMD（NEON）优化

x86 与 ARM 对比以及其他对比分析

- x86 与 ARM 程序性能差异
- x86 与 ARM 列划分差异
- x86 平台的程序在 1024 维度性能更好

(二) OpenMP

x86 平台

- 线程数对 OpenMP 优化的影响
- OpenMP 结合 SIMD（AVX、SSE）

ARM 平台

- 调度策略与 chunk_size
- 行划分与列划分对比

x86 与 ARM 对比以及其他对比分析

- x86 与 ARM 程序性能差异

(三) Pthread vs OpenMP

- 分析 Pthread 和 OpenMP 两种并行编程方式的程序性能差异

七、 Pthread 实验结果分析

(一) x86 平台下的实验结果分析

1. 三种静态线程优化策略对比

对“Pthread 设计实现”部分的三种静态线程优化策略进行测试与分析，选取线程数为 8 时的数据。

表 1: 三种静态线程优化策略执行时间对比（单位：μs）

问题规模	串行	静态 + 信号量	静态 + 信号量 + 三重循环	静态 +barrier
32	40	2422	1722	1504
64	325	4352	2916	2308
128	2188	8311	5383	4054
256	15568	19282	11714	7573
512	140059	54636	27668	21054
1024	1139062	137243	85605	67265
2048	9196672	1326324	1445745	1389486

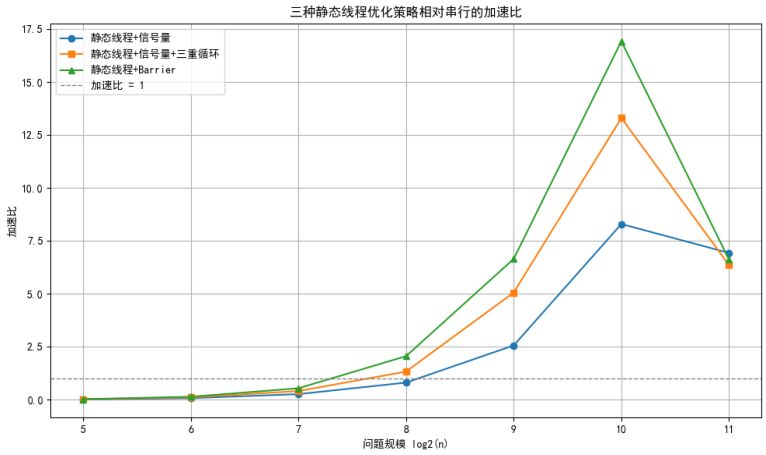

图 1: 三种静态线程优化策略相对串行的加速比

从表格与加速比曲线图中可以看出，随着问题规模的增大，三种 `Pthread` 静态线程优化策略相较于串行版本在性能上有显著提升。对于小规模（$n \leq 128$）的问题，由于线程创建与同步的额外开销，并行策略的开销反而更大。但是随着矩阵规模扩大（$n \geq 256$），三种优化策略的优势开始显现。其中，“静态线程 + 信号量同步”策略在 $n = 512$ 后取得了明显的加速效果，而“静态线程 + `barrier` 同步”策略由于同步结构更加简洁高效，在 $n = 512$ 至 $n = 1024$ 范围内加速效果最佳，最高达到了约 17 倍的加速比，表现出优越的负载均衡能力和同步效率。同时，“静态线程 + 信号量同步 + 三重循环全部纳入线程函数”的策略在中等规模 ($n = 256$ 到 $n = 1024$) 下兼顾线程利用率与同步开销，展现出良好的扩展性和计算效率。综上所述，三种静态线程优化策略在问题规模较大时都显著优于串行算法，其中 `barrier` 同步策略综合性能最优，所以合理的线程划分与同步机制对于提升并行高斯消元性能至关重要。

2. 静态优化策略不同线程数对比

为研究线程数量对并行效率的影响，我们分别在 $num\_Threads = 2, 4, 8$ 的线程配置下，测试了三种静态线程优化策略的性能表现，并计算了各策略在不同线程数下相对于串行算法的加速比，进行对比分析。

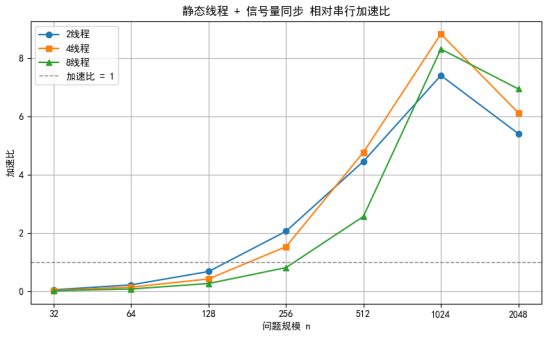

图 2: 静态线程 + 信号量同步串行加速比

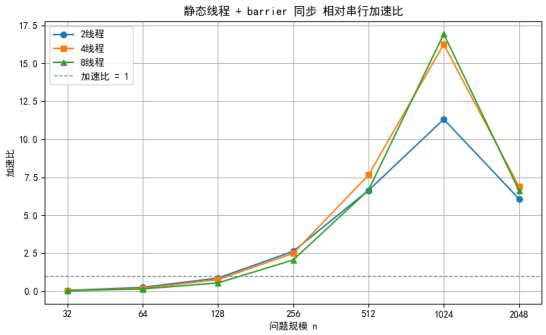

图 3: 静态线程 + Barrier 同步串行加速比

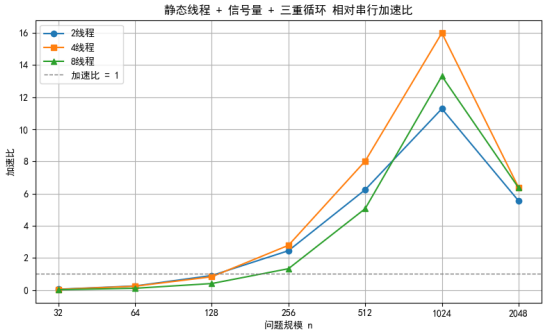

图 4: 静态线程 + 信号量 + 三重循环加速比

- 整体来说，无论线程数多少，信号量同步优化相较于另外两个策略，性能较差。
- 当线程数从 2 增加到 4 时，所有策略均出现明显性能提升，在线程数为 4 时性能较好，并行度提高带来了更高的吞吐能力，扩展性好；但是当进一步提升至 8 线程时，整体性能反而下降，此时线程同步开销和等待代价显著增长，成为主要性能瓶颈，尤其在信号量同步策略中体现得最为明显。
- 在信号量同步和三重循环两个策略的 8 线程下，问题规模较小时，部分线程可能因任务较少而处于空闲或频繁等待状态，导致并行资源未被充分利用，性能下降十分明显。

- 尽管主消元过程是并行的，但是每轮的除法操作（主线程对第 k 行的标准化处理）是串行的。随着线程数提升，串行部分比重愈发明显，成为制约程序性能重要因素。
- 与信号量相比，barrier 机制在多线程环境下具有更低的同步开销，其原子性和集合等待特性避免了多个信号量的逐一协调。所以 4 和 8 线程下，barrier 同步策略更能体现出同步的高效性和可扩展性，性能表现最为稳定最优。

综上所述，性能受限于同步机制、线程管理以及串行段比例等多个因素，只是简单地增加线程数并不会线性提升性能，反而可能引入额外开销，应当合理控制线程数量并选择适合的同步机制。

3. 静态线程与动态线程对比

我选取静态线程中整体性能最好的 barrier 同步策略与动态线程进行对比分析。

表 2: 静态线程（barrier 同步）与动态线程的执行时间对比（单位：μs）

问题规模	静态线程	动态线程
32	1504	27928
64	2308	110316
128	4054	430543
256	7573	1785005
512	21054	8359920
1024	67265	37027523
2048	1389486	—

从表格中可以看出，静态线程在整体性能上极大地优于动态线程，并且动态线程相对于串行算法来说甚至是极大的负优化。因为动态线程有着非常频繁的线程创建和销毁开销，代码量少是其唯一优点了。

4. 不同任务划分策略（行/列划分）对比

为了研究不同的任务划分策略对高斯消元性能的影响，我选取综合性能最好的静态线程+barrier 同步优化策略，分别对消去阶段采用 **行划分**和 **列划分**进行对比实验。

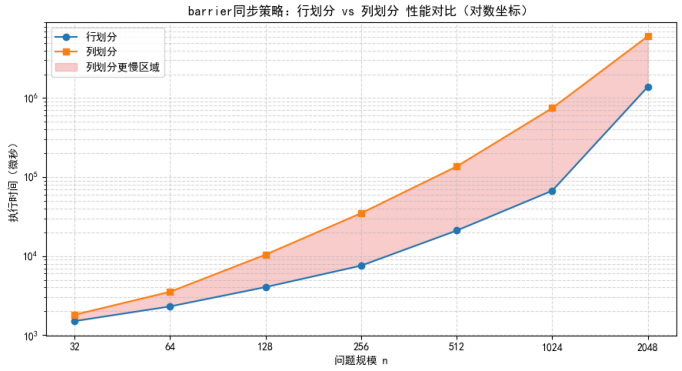

图 5: barrier 同步：行划分 vs 列划分

- 在所有问题规模下，**行划分策略的性能始终优于列划分策略**，并且随着问题规模的增大，行划分优势逐渐显著。

- 列划分方式将每一行中不同列的更新分给多个线程处理，导致多个线程需要频繁访问和写入同一行的数据，引发 **写入冲突**和 **Cache Line 共享（伪共享）**问题，从而造成严重的性能退化。
- 相比较而言，行划分将整行任务交由不同线程独立完成，每个线程更新的内存区域相互独立，从而能更好地利用缓存，避免写竞争和数据同步冲突，具有更优的内存访问局部性。
- 另一方面，行划分在实现上更简单，不需要对每行加锁或使用原子操作进行同步，也因此更适合在 barrier 同步模式下的粗粒度并行。

综上所述，行划分不仅性能更好，并且在大规模问题中更具扩展性和可维护性。而列划分由于线程管理和读写数据的问题，性能远不及行划分。

(二) ARM 平台下的实验结果分析

1. 三种静态线程优化策略对比

在华为鲲鹏服务器上对三种静态线程优化策略进行测试与分析，选取线程数为 8 时的数据。

表 3: 三种静态线程优化策略执行时间对比（单位：μs）

问题规模	串行	静态 + 信号量	静态 + 信号量 + 三重循环	静态 +barrier
32	74	3567	2715	2839
64	573	6300	4971	4950
128	4510	11349	13177	9334
256	35565	28746	25942	26307
512	293934	82048	78509	91084
1024	2415255	454512	389754	291638
2048	21009373	2797632	2638836	2032767

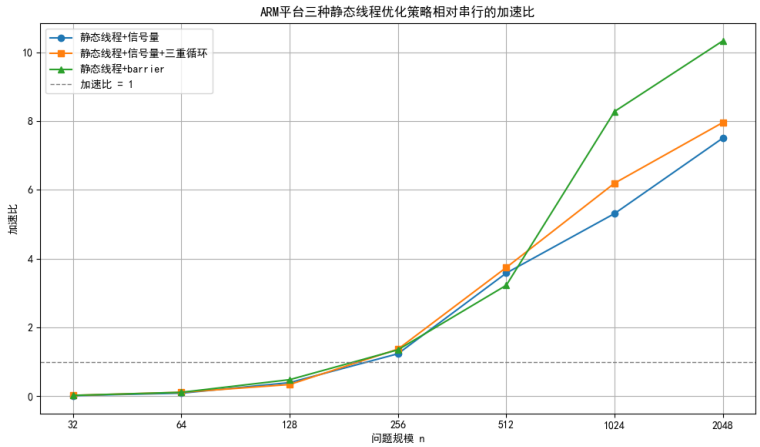

图 6: ARM 平台三种静态线程优化策略的加速比

- 整体来看，三种优化策略都有着不错的性能，并且随着问题规模的增大，三种优化策略的加速比都在上升。

- “静态线程 + 信号量同步” 由于信号量操作开销和线程调度不够高效，导致优化力度相较于其他两种优化策略稍稍落后。
- 在小规模问题（$n \leq 128$）下，三种策略均存在明显的性能退化，这主要是因为线程创建、调度和同步的额外开销远高于并行计算所带来的收益。
- 当问题规模增至 $n \geq 256$ 后，加速比逐渐提升。其中 “静态线程 + barrier 同步” 策略在 $n = 512$ 后开始明显优于其他策略，至 $n = 2048$ 时取得最高加速比约为 10.34，显示出优越的可扩展性和同步效率。

2. 静态优化策略不同线程数对比

在 ARM 平台上研究线程数量对并行效率的影响，分别在 $num\_Threads = 2, 4, 8$ 的线程配置下，测试三种静态线程优化策略的性能表现，并计算各策略在不同线程数下相对于串行算法的加速比，进行对比分析。

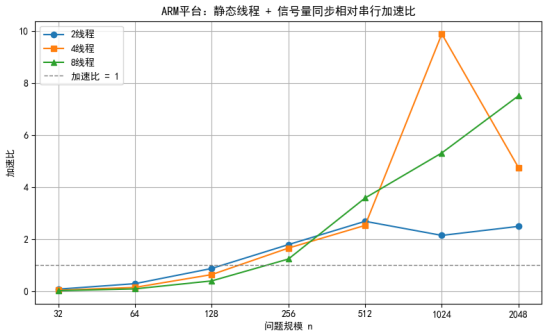

图 7: ARM 静态线程 + 信号量同步加速比

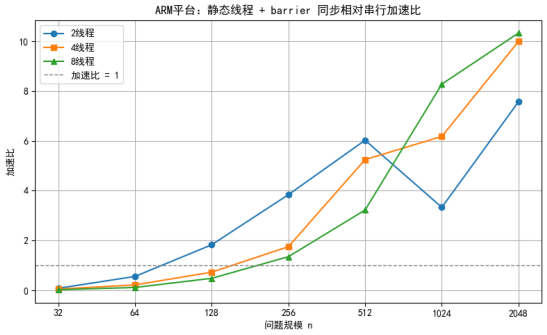

图 8: ARM 静态线程 + barrier 同步加速比

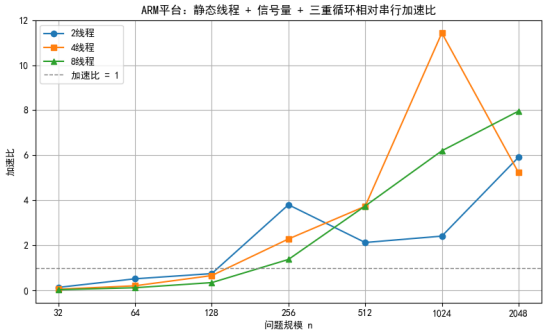

图 9: ARM 静态 + 信号量 + 三重循环加速比

通过加速比曲线图可以观察到，随着线程数从 2 增加到 4，所有策略在 $n \geq 128$ 的加速效果均显著提升，说明 ARM 架构在多线程并行计算上具备一定的扩展能力，尤其对于计算量较大的任务，其并发性可以被充分利用。其次，与 x86 平台一样，三种策略在小规模问题上，由于线程的创建和销毁开销远高于并行计算所带来的收益，性能反而不如串行算法。而且线程数并不是越多越好，三种策略整体上都是在线程数为 4 时性能最优。此外，三种优化策略中，“静态线程 + barrier 同步” 整体加速表现最优，具有更好的同步效率和负载均衡能力。

与 x86 平台不同的是，ARM 平台上线程数为 2 时，三种优化策略的性能都与 4 和 8 线程有比较大的差距，分析原因可能是 ARM 架构下核心间资源隔离较强，线程调度和负载均衡机制对小规模并发不够敏感，容易导致线程空闲或不均衡的现象发生。此外，由于 ARM 处理器每个核心仅支持单线程，无法像 x86 的 SMT 架构一样通过硬件线程重叠隐藏延迟，因此当仅有两个线程参与计算时，CPU 资源利用率偏低，且 barrier 或信号量的同步开销相对于计算时间占比较高，进一步降低了整体加速比。随着线程数增加到 4 或 8，线程间的工作划分更加充分，调度效率提高，性能才得以明显改善。

3. Pthread&NEON 优化

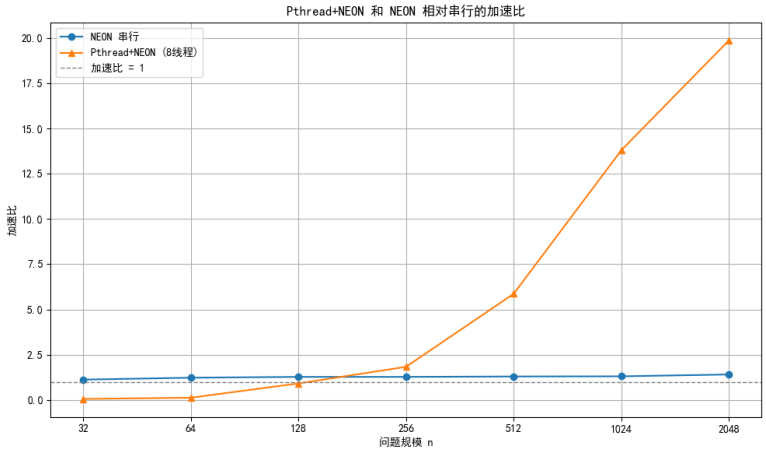

图 10: Pthread+NEON 和 NEON 相对串行的加速比

从图中可以看出，NEON 向量化优化策略在 ARM 平台的加速比始终稳定在 1.0 ∼ 1.4 左右，其性能优势主要体现在对单线程执行过程中浮点运算的加速，但未利用多核资源，因此随着问题规模扩大，其加速效果提升有限。相比之下，Pthread + NEON（8 线程）策略在问题规模较小（$n \leq 128$）时因线程调度和同步开销较大，性能反而不如 NEON；而当问题规模增加至 $n \geq 256$ 时，Pthread+NEON 利用多线程并行执行的优势逐渐显现，在 $n = 2048$ 时加速比高达 20 倍以上，显著优于 NEON 单线程优化。

造成这种现象的主要原因在于：NEON 本质上是数据级并行（SIMD），适合加速向量内连续的数据操作，但仍局限于单核执行，无法跨核心扩展；而 Pthread 实现的是任务级并行（MIMD），能充分利用多个核心同时执行矩阵消元任务。随着矩阵规模扩大，任务粒度变大，线程之间的同步成本相对下降，因而加速比提升更为明显。综合而言，在大规模问题中，Pthread+NEON 的组合策略能够最大限度地发挥 ARM 平台多核和 SIMD 指令集的联合优势，具有更好的扩展性和性能表现。

(三) x86 与 ARM 对比以及其他对比分析

1. x86 与 ARM 程序性能差异

将 Pthread 并行程序从 x86 平台迁移至 ARM 平台后，在相同问题规模下，ARM 平台的运行时间整体上要显著高于 x86 平台。分析造成这一现象的原因主要包括以下几点：

- **单核性能差异显著：** Kunpeng-920 为服务器架构，采用低主频高并发设计，BogoMIPS 仅为 200.00，而 AMD Ryzen 5800H 主频高达 3.2 GHz，具备较强的单核浮点性能。在串行部分占比较高的高斯消元中，单核性能直接影响整体计算速度。
- **无超线程支持：** Kunpeng-920 每核仅支持 **1 个线程**，而 Ryzen 5800H 拥有 **8 核 16 线程**，可以利用 SMT（Simultaneous Multithreading）技术提高并发执行效率，线程调度更灵活。
- **编译器优化与库支持不足：** x86 平台的编译器和优化工具链更为成熟，在 Pthread、OpenMP 等并行编程库的调度和同步方面更优化；而 ARM 架构尤其是鲲鹏平台上，线程调度、内存模型等支持可能相对不够充分，影响多线程执行效率。
- **同步机制与系统调度策略差异：** Pthread 中使用的 `barrier` 和 `semaphore` 等同步原语在不同平台的系统调用实现开销存在差异，ARM 系统可能在这些原语的效率上略逊于 x86 系统，导致并行效率不如预期。

2. x86 与 ARM 列划分差异

x86 平台和 ARM 平台下的静态线程与动态线程对比类似，趋势以及现象没有太大差异。但是我在 ARM 平台进行不同任务划分策略对比时，发现列划分程序执行时间波动非常大，性能并不稳定，而 x86 平台中没有这个问题。

分析造成 ARM 与 x86 平台在列划分策略下性能波动差异的核心原因在于两者缓存架构和访问机制的显著不同。列划分会导致频繁的跨行访问，而现代内存按行连续布局，使得列访问在缓存友好性上处于劣势。在 ARM 平台上，尽管其拥有高达 48MB 的 L3 缓存，但由于其采用分布式共享架构（DSU），不同核心访问需通过互联总线协调，访问延迟更高、竞争更激烈，特别容易在数据交叉访问中触发缓存抖动与 Cache Thrashing。此外，其 L1 缓存较小（128KB）且未支持 SMT，多线程并发调度灵活性较低，进一步放大了列访问带来的主存访问压力。相比之下，x86 平台的 AMD Ryzen 7 5800H 具有更优化的共享 L3 缓存（16MB）、完备的缓存一致性协议以及高效的数据预取机制，使其在列优先访问模式下表现更为稳定。

我觉得程序执行时间波动大，还可能与随机值有关，因为矩阵的元素是随机生成的，列划分可能更适合适合并行度高但每行数据小的情况。

3. x86 平台的程序在 1024 维度性能更好

我还发现一个有意思的事情，就是 x86 平台下，我所有的 Pthread 程序在问题规模为 1024 时的加速比是最高的，也就是说优化力度是最大的。这种现象的原因可以从计算密集度与资源利用效率两方面进行解释。一方面，当问题规模较小时（$n = 32 \sim 256$），任务粒度较小，线程创建与调度开销相对占比更高，导致并行计算未能充分发挥性能潜力；而当 n 增大至 1024 时，每个线程的任务量足够大，从而有效摊销了线程管理与同步的开销，达到并行效率的最优区间。另一方面，$n = 1024$ 时内存访问模式仍然能较好地维持在 CPU 高速缓存（L2/L3）范围内，避免了频繁的主存访问瓶颈，使得数据局部性与 cache 利用效率最大化。然而，继续增大问题规模（$n = 2048$），尽管线程工作负载进一步增多，但数据量的增长会导致 cache 未命中率显著上升，

线程间同步频率增加，甚至触发内存带宽瓶颈，从而降低了整体加速效果。因此，$n = 1024$ 成为了计算密集与资源调度之间最优平衡点，是线程并行策略发挥最大优化力度的临界规模。

八、 OpenMP 实验结果分析

(一) x86 平台下的实验结果分析

1. 线程数对 OpenMP 优化的影响

对 OpenMP 不同线程数（2、4、8）在不同问题规模下的运行时间与加速比结果进行分析。

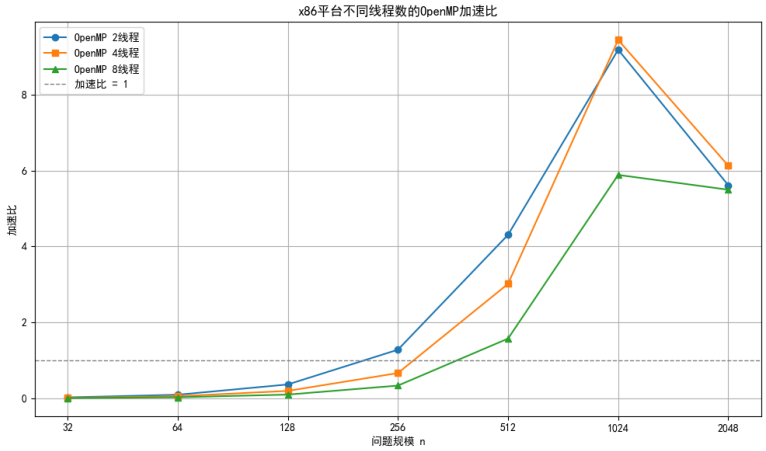

图 11: x86 平台不同线程数的 OpenMP 加速比

整体上，2 线程的加速效果要优于 4 线程优于 8 线程，性能最好。在问题规模 $n \leq 128$ 时，加速比显著低于 1，性能较差。随着问题规模增大，加速比显著提升，并且 4 线程在处理大规模问题时表现更好。上述现象可以从以下几个方面进行深入分析：

- **线程启动与调度开销**：在小规模问题中，实际参与计算的数据量有限，OpenMP 启动多个线程所带来的额外调度与同步开销超过了并行带来的收益，导致总体性能下降。
- **负载不均与线程饥饿**：在高斯消元的算法结构中，每一轮的消元操作随着 k 的增大而缩减计算规模，这种动态变化在默认的 `static` 分配策略下容易造成线程空闲与负载不均现象。特别是在 8 线程并行中，更容易出现部分线程无事可做的“线程饥饿”问题，从而影响整体性能。
- **共享内存带宽与缓存争用**：随着线程数量的增加，多个线程同时访问共享矩阵数据，容易造成内存带宽饱和和缓存竞争。尤其在 8 线程的情况下，线程间争夺 L3 缓存资源更为严重，带来访存延迟，从而抵消了部分并行计算带来的速度提升。
- **NUMA 架构与线程亲和性**：在多核多线程的 x86 平台中，每个线程所在的核心与内存控制器之间存在访问延迟差异，即 NUMA 特性。当 OpenMP 没有显式绑定线程亲和性时，调度器可能将线程迁移到远核，从而造成频繁的数据搬移和性能抖动。

综上所述，OpenMP 的并行性能受线程数、任务粒度、内存结构与调度策略等多因素影响。在 x86 平台和本实验实现结构下，2 线程在多数规模上表现最优，取得了明显加速效果。而线程数的增加并非始终带来线性加速，尤其在资源受限或调度不合理的情况下，8 线程性能不升反降。

2. OpenMP 结合 SIMD（AVX、SSE）

将 OpenMP 分别与 AVX 和 SSE 结合，采用性能较好的 2 线程，与单纯的 AVX 和 SSE 对比分析。

表 4: SIMD 与 OpenMP+SIMD 性能对比实验数据（单位：μs）

问题规模	AVX	AVX+OpenMP（2 线程）	SSE	SSE+OpenMP（2 线程）
32	3	1720	3	1856
64	22	3233	13	3170
128	140	6744	102	6470
256	1161	12903	799	13077
512	8519	31971	5194	31535
1024	67947	100016	45495	83839
2048	1573614	1209214	708196	411627

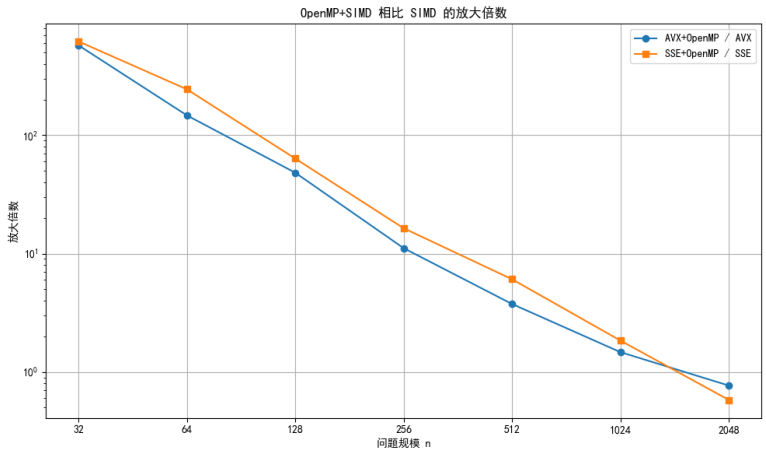

图 12: OpenMP+SIMD 相比 SIMD 的放大倍数

从实验数据与图像中可以明显观察到，在 x86 平台上，无论使用 AVX 还是 SSE 指令集，当与 OpenMP 的多线程并行结合后，执行时间均显著增加，尤其在问题规模较小时更为明显。OpenMP 多线程与 SIMD 向量化虽然理论上可以叠加提速，但在实际实现中却可能由于调度开销、数据对齐失效、缓存干扰等系统性因素而互相干扰，尤其在计算任务较轻时，OpenMP 的引入可能反而成为性能瓶颈。这一现象违背了直觉上的“线程数增加应加速”的假设，其本质原因可从以下几个方面进行分析：

- **线程调度与同步开销显著**：OpenMP 在多线程并行过程中会引入线程创建、调度、上下文切换以及同步的额外开销。当问题规模较小时，实际的计算任务本身较轻，线程间的调度和同步成本反而成为主要瓶颈，掩盖了 SIMD 并行本身的加速效果。
- **数据划分破坏向量加载对齐性**：AVX 和 SSE 指令对内存访问的对齐性要求较高。在单线程 SIMD 程序中，内存通常连续，易于形成对齐访问，而引入 OpenMP 多线程后，数据可能被分割至不同线程处理，增加了非对齐访问的概率，导致额外的内存加载延迟，降低了指令吞吐效率。

- **缓存竞争与伪共享问题**：多个线程并发访问共享的内存空间，尤其是矩阵与向量的数据结构，会引发 L1/L2/L3 缓存的竞争，甚至出现伪共享现象。这不仅会导致缓存一致性协议带来的通信延迟，还会进一步影响 SIMD 的高效执行。
- **SMT 线程调度干扰 SIMD 指令管线**：在 x86 架构中，支持 SMT，一个物理核心可能执行多个线程，但这会导致 SIMD 指令管线与其他线程产生资源争夺，从而影响 AVX/SSE 指令的并行吞吐能力，反而拖慢整体执行速度。

(二) ARM 平台下的实验结果分析

1. 行划分与列划分对比

我在 x86 平台和 ARM 平台都分别实现了基于行划分和列划分的 OpenMP 并行优化方案：

- **行划分**：使用 `#pragma omp for` 对 i 行进行并行处理，每个线程负责若干行的消去更新。这种方式结构清晰，负载均衡性较好，适用于大多数平台。
- **列划分**：对内层 j 列进行并行划分，每个线程更新若干列。这种方式可以增强列向量访问的并行性，但因现代内存通常按行存储，若处理不当可能会带来缓存局部性问题。

在这里我主要对 ARM 平台的实验结果（8 线程）进行分析，并对比说明 x86 平台的不同。

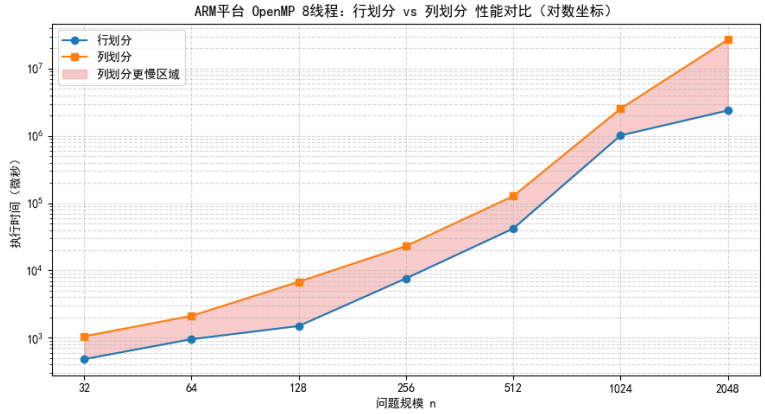

图 13: ARM 平台 OpenMP 8 线程：行划分 vs 列划分执行时间对比

图中可以看到，无论问题规模为多少，行划分始终是要比列划分性能要好的。这主要归因于现代处理器普遍采用按行优先的内存存储方式，行划分可以更好地利用数据的空间局部性和缓存预取机制。而列划分则涉及大量跨行访问，容易导致缓存失效和内存带宽压力，尤其在计算规模扩大后，跨行访问对缓存层次结构造成更大负担，进而降低整体性能。因此在共享内存架构下，行划分更具内存友好性，是更适合高斯消元的并行划分方式。

与 ARM 平台不同的是，x86 平台上列划分与行划分的性能差距要更大，执行时间大约在行划分的 3 倍，这一差异可归因于 x86 处理器对缓存和内存层次的管理更加精细，具有较大的共享 L3 缓存，但对数据访问模式也更为敏感。列划分策略会导致多个线程同时访问不连续的内存地址，产生频繁的缓存行替换与冲突，显著增加主存访问次数。此外，由于 x86 平台线程数量更多，列划分造成的带宽竞争和总线压力也更加突出，从而进一步放大了与行划分之间的性能差距。

2. 调度策略与 chunk_size

我在 ARM 平台通过设置 `chunk_size` 为 $1, 4, 8, \ldots, 256$ 等多个级别，分别测试 static、dynamic 和 guided 结果如下图。

static	执行时间							
32	482							
64	954							
128	1499							
256	7648							
512	42122							
1024	302546							
2048	2385961							
dynamic	1	4	8	16	32	64	128	256
32	207	186	208	250	266	286	253	258
64	626	556	606	742	963	1185	1263	1208
128	1895	1522	1574	1801	2751	4263	5942	5864
256	14195	11854	11850	12972	16145	25332	41154	55678
512	65271	59207	59481	63110	72262	98607	190380	309181
1024	438679	416399	423255	409829	452732	524854	787350	1258939
2048	3359415	3193975	3292164	3285854	3409441	3668759	4381140	5491943
guided	1	4	8	16	32	64	128	256
32	215	182	194	227	243	241	259	260
64	577	524	571	698	874	1092	1057	1026
128	1587	1478	1567	1765	2649	4145	5706	5602
256	11937	11148	12088	13374	17150	27353	49558	78497
512	59456	59976	59405	62616	72836	98094	189318	309131
1024	393302	397529	397337	465199	481438	519773	684336	1325133
2048	3300538	3349214	3378662	3983277	4466950	4632295	5162045	6081712

图 14: static、dynamic、guided 在不同 chunk_size 下的结果

`static` 策略采用固定块大小的任务划分方式，由 OpenMP 根据线程数自动划分任务，将循环迭代任务平均分配至各线程，调度开销最小。测试结果表明，对于中等及大规模问题（$n \geq 256$），`static` 策略整体性能较为稳定并且性能要始终优于 dynamic 和 guided。

`dynamic` 策略按需将 chunk_size 大小的任务动态分配给空闲线程，具备更强的负载均衡能力。实验数据显示，`dynamic` 策略在所有规模下都表现出了良好的自适应性，尤其在中等规模问题（$n = 128 \sim 1024$）上，当 chunk_size 设置为 4 或 8 时达到最优性能；而当 chunk_size 过大（如 64、128、256）时，由于单个线程持有任务粒度过大，调度灵活性下降，线程间负载再平衡能力减弱，导致执行时间显著上升。此外，过小的 chunk_size（如 1）会带来频繁的调度开销，在问题规模较大时同样导致性能下降。

`guided` 策略采用指数下降的方式分配任务，初期分配较大 chunk，随后逐步减小，兼顾了初期并行度和后期调度精细度。实验结果表明，`guided` 策略整体趋势与 `dynamic` 类似，在 $n = 128 \sim 1024$ 范围内 chunk_size 为 $4 \sim 16$ 时性能较优。相比 `dynamic` 策略，`guided` 在大规模计算中具有更平滑的性能下降趋势，表明其在任务负载逐步减少的迭代算法中更具鲁棒性。但在极端小或极端大的 chunk_size 下，同样面临调度频繁或负载不均的问题。

虽然 `dynamic` 与 `guided` 策略在一般负载不均的循环结构中具备更强的自适应能力，但对于高斯消元问题而言，`static` 策略往往能取得更优性能。这是因为在每轮消元过程中，对于同一轮而言，各线程执行的任务量是高度均匀的，几乎不存在线程负载显著不平衡的情况。因此，`static` 策略在一次性将任务平均划分至各线程后，能够显著减少调度开销，避免频繁的任务分派和线程抢占，从而更好地发挥 CPU 缓存与流水线并行效率。相反，`dynamic` 与 `guided` 策略的调度机制虽然具备弹性，但其额外的调度开销在负载均衡收益不显著的场景中反而成为性能瓶颈。因此，在高斯消元这类结构规整、阶段负载均匀的问题中，`static` 策略通常更具优势。

(三) x86 与 ARM 对比以及其他对比分析

1. x86 与 ARM 程序性能差异

与 Pthread 不同，OpenMP 的 x86 与 ARM 程序性能差异在于：ARM 平台的程序性能在问题规模小时优于 x86，而问题规模大时 x86 平台程序性能更好。

表 5: OpenMP 不同线程数在 x86 与 ARM 平台的执行时间对比（单位：μs）

问题规模	x86 平台			ARM 平台		
	openmp2	openmp4	openmp8	openmp2	openmp4	openmp8
32	1694	3193	5945	239	394	482
64	3451	6010	11808	497	754	954
128	5992	11189	22320	2529	2204	1499
256	12155	23393	46459	19199	11164	7648
512	32497	46415	88968	145379	75069	42122
1024	123953	120592	193422	1156866	580655	1017428
2048	1637803	1500110	1672463	9418961	4577917	2385961

造成上述现象的根本原因在于两种架构在硬件资源与体系结构方面的差异，具体分析如下：

- **一级缓存容量对小规模问题的影响**：Kunpeng-920 每核配备 **128 KB 的 L1 Cache**，而 AMD Ryzen 7 5800H 每核仅有 **64 KB**，因此在处理小规模问题时，ARM 平台更能容纳完整的数据矩阵与中间变量，减少访存次数，提升缓存命中率，使得并行计算受益显著。
- **主频与计算密度对大规模问题的影响**：Ryzen 7 5800H 拥有**更高的主频**，并支持 **每核 SMT 超线程（共 16 线程）**，这使得在处理大量计算任务时具备更强的计算能力和线程调度效率。大规模高斯消元（如 $n = 1024$、$n = 2048$）中的浮点运算和内存访问量剧增，x86 平台因其高频核心与成熟的流水线执行机制，在这种计算密集型场景中具备明显优势。
- **三级缓存容量对大规模并行的支持**：虽然 Kunpeng-920 配备了 **48MB 的 L3 Cache**，表面上优于 x86 的 16MB，但其访问延迟更高、核心间共享机制较弱，在 OpenMP 并行中若线程频繁访问共享数据，反而可能因缓存一致性开销带来性能瓶颈。而 Ryzen 的 L3 Cache 结构更加优化，且具备更强的跨核心数据一致性能力，有助于大规模任务下保持高效协同。

综上所述，ARM 平台在处理小规模问题时受益于更大的 L1 缓存与高效的内存访问路径，因此性能更优；而在问题规模扩大后，x86 平台凭借高主频、强 SIMD 能力及更优线程调度机制，在整体计算能力和缓存协调方面更具优势，从而实现更好的并行加速效果。

九、 Pthread&OpenMP 对比分析

分别对 Pthread 和 OpenMP 进行多方面探索之后，接下来，我们来对这两种并行编程方法进行对比分析，数据如下表所示，Pthread 所选数据为性能较好的静态线程（barrier）线程数为 8 时的数据，OpenMP 也是线程数为 8 时的数据。

表 6: x86 与 ARM 平台下串行、Pthread、OpenMP 执行时间对比（单位：μs）

问题规模	x86 平台			ARM 平台		
	串行	Pthread	OpenMP	串行	Pthread	OpenMP
32	40	1504	5945	74	2839	482
64	325	2308	11808	573	4950	954
128	2188	4054	22320	4510	9334	1499
256	15568	7573	46459	35565	26307	7648
512	140059	21054	88968	293934	91084	42122
1024	1139062	67265	193422	2415255	291638	1017428
2048	9196672	1389486	1672463	21009373	2032767	2385961

x86 平台表现分析 在 x86 平台上，Pthread 的性能在各个规模下均优于 OpenMP。这一现象主要源于 Pthread 更加灵活的线程控制与同步机制。通过预先创建固定数量的工作线程并配合 `pthread_barrier` 同步，Pthread 避免了每轮迭代中线程的反复创建与销毁，线程生命周期覆盖整个消元过程，极大降低了调度开销。此外，x86 架构支持超线程以及较为成熟的多级缓存一致性机制，Pthread 可以结合线程亲和性（CPU affinity）精确绑定核心，最大化缓存命中与并行性。而 OpenMP 采用基于指令注释的高层抽象，在并行区域的线程调度、工作划分由编译器和运行时系统自动完成，虽然便于开发，但容易在复杂控制逻辑下引入冗余同步或负载不均等问题，尤其在高斯消元这种任务规模随轮次动态变化的算法中，固定划分可能导致部分线程空闲。

ARM 平台表现分析 在 ARM 平台上,两者性能差距相对缩小,甚至在问题规模较小时,OpenMP 反而优于 Pthread。这一现象与 ARM 的硬件和操作系统线程模型有关。一方面，ARM 服务器架构中系统调用开销相对更大，Pthread 中涉及的 `pthread_create`、`join`、`barrier_wait` 等接口在执行频繁同步时带来更高代价；另一方面，OpenMP 编译器在并行区块中使用轻量级线程池和线程复用机制，使其在小规模任务中能更快进入并行状态。此外，ARM Kunpeng-920 平台拥有更大的 L1 数据缓存，对短小向量或小矩阵操作更加友好，OpenMP 的默认调度策略结合编译器自动向量化优化，在小规模问题中能够充分发挥体系结构的高速缓存能力。然而，随着问题规模扩大（$n \geq 1024$），任务计算量上升且同步变得频繁，OpenMP 线程间调度机制开始暴露出调度延迟、负载不均等问题，而 Pthread 因其同步逻辑由用户手动控制，且线程绑定明确，反而能够稳定地维持较高的并行效率。

综合比较与结论 总体来看，**Pthread 适用于对并行性能有较高要求、可接受开发复杂度的大规模密集计算任务**，其显著特点在于高控制力、可配置性强，特别适合高斯消元这类计算结构清晰、同步时序明确的算法。在程序调度、线程绑定和缓存利用方面均可进行精细优化。而 **OpenMP 更适合开发周期短、结构规则的中小规模并行任务**，其编程模型更贴近串行逻辑，仅需添加编译指令即可实现并行化。其劣势在于调度机制难以微调，对于计算模式复杂或具有阶段性任务变化的应用，其并行效率受限。

十、 遇到的问题以及说明

- 在 ARM 平台进行数据测量时，有时会发现数据波动很大。而且对编译选项的设置并不熟悉，尤其是在与 SIMD 结合时，编译选项需要都开启。
- 因为改变了矩阵的初始化方式，所以我将需要用到的部分 SIMD 实验的数据重新测量了。

十一、 总结

通过本次并行编程实验，深入对比了 Pthread 与 OpenMP 两种主流并行模型在高斯消元算法中的应用与性能差异。在实验过程中，不仅掌握了线程创建、同步、任务划分等并行编程核心技术，加深了我对计算架构与并行策略背后原理的理解，也体会到了不同并行策略在不同平台、不同规模下的适用性与优化空间。

源代码链接Github