南開大學
Nankai University

计算机学院

并行程序设计实验报告

pthread 以及 OpenMP 编程

姓名：张明昆

学号：2211585

专业：计算机科学与技术

2024 年 5 月 26 日

目录

1 引言

1.1 课题背景及意义

高斯消去法是一种经典的线性代数算法，用于解线性方程组。该方法通过一系列初等行变换将矩阵化为上三角矩阵，从而简化解的过程。高斯消去法广泛应用于科学计算、工程分析、经济模型等领域。然而，随着数据规模的增大和计算需求的增加，单线程的高斯消去法逐渐显现出性能瓶颈。

多线程并行计算通过同时利用多个处理器核心，能够显著提高计算效率。特别是在现代计算中，多核处理器已成为主流硬件配置，充分利用多线程并行计算的潜力，对提升高斯消去法的性能具有重要意义。

1.2 目标和要求

本次作业的目标是使用 pthread 和 OpenMP 对高斯消去法进行并行化实验，并结合 SIMD（单指令多数据流）优化，进一步提升算法性能。具体要求包括：

- 设计并实现适合的任务分配算法，分析其性能。
- 将高斯消去法与 SIMD（如 Neon、SSE/AVX/AVX-512）算法相结合。
- 在 ARM 平台或 x86 平台上编程实现，并测试不同问题规模、不同线程数下的算法性能。
- 比较串行和并行算法的性能差异，讨论 pthread 程序和 OpenMP 程序的性能表现。
- 探索 pthread 和 OpenMP 并行化特殊高斯消去法。
- 在不同平台上（如 x86 或 ARM）进行普通及特殊高斯消去法的并行化实验。
- 讨论多线程并行化的不同算法策略（如矩阵水平划分、垂直划分等），保证不同算法策略下的一致性。
- 进行复杂性分析，包括 profiling 及体系结构相关优化（如 cache 优化）。
- 对 OpenMP 卸载到加速器设备以及与 oneAPI 编程的性能进行比较。

2 理论基础

2.1 高斯消去法简介

高斯消去法是一种用于求解线性方程组的算法。其基本原理是通过初等行变换将矩阵化为上三角矩阵，然后通过回代求解。高斯消去法的步骤包括消元和回代两个阶段。该算法的时间复杂度为 $O(n^3)$，其中 n 为矩阵的阶数。

2.2 SIMD 的基本概念

SIMD（Single Instruction, Multiple Data）模式是一种并行计算技术，允许单条指令同时作用于多个数据点。SIMD 在处理向量和矩阵运算、图形处理和多媒体应用中具有重要作用，通过并行处理提高计算效率。

2.3 pthread 的基本概念

POSIX 线程库（pthread）是 POSIX 标准的线程库，用于多线程编程。pthread 提供了一组 API 函数，用于创建、控制和管理线程。通过 pthread，程序可以并行执行不同的任务，从而提高程序的执行效率。

2.4 OpenMP 的基本概念

OpenMP 是一种用于多平台共享内存并行编程的 API，提供了丰富的编译器指令、库函数和环境变量。OpenMP 通过简化的编程模型，使得程序员可以轻松地将串行代码并行化，以充分利用多核处理器的性能。

2.5 ARM 与 x86 平台简介

ARM 和 x86 是两种不同的处理器架构。ARM 架构以其低功耗和高效能见长，广泛应用于移动设备和嵌入式系统；x86 架构则以其强大的计算能力和丰富的软件生态著称，主要用于个人计算机和服务器。两者在设计理念和应用场景上有显著差异。

3 算法设计

3.1 任务分配算法及其性能分析

为了实现并行化，可以按以下步骤进行设计：

1. 使用单个线程进行消元行的除法操作，确保消元行的主元变为 1。
2. 消元行除法操作完成后，使用多个线程并行处理被消元行的减法操作。

3.1.1 pthread 数据块划分设计

在任务划分时，传统方法采用等步长划分方式，这种划分方式存在一定的弊端。当数据规模较大时，由于 L1 cache 大小有限，可能会导致在访问下一个间隔为线程数的行时出现 cache miss，从而需要到 L2、L3 甚至内存中读取数据，造成额外的访存开销。因此，考虑采用一种充分利用 cache 优化的数据划分方式，即将数据按块划分。每个线程负责连续的几行消去任务。这样做的好处是，当线程正在处理当前行时，CPU 可能会提前预取下一行的数据到 cache 中，从而减少不必要的访存开销。

3.1.2 pthread 数据动态划分设计

在任务划分时，由于不同线程执行任务所需时间可能不一致，甚至因为数据规模不是线程数量的整数倍，某些线程在个别轮次中会出现空等待的状态。这是由于数据划分时细粒度的数据划分导致的线程之间负载不均衡。

为了解决这一问题，考虑采用动态数据划分方式。在对被消元行执行减法操作时，并不明确指定某个线程对哪部分数据执行任务，而是根据各个线程任务完成的情况动态地进行数据划分。通过一个全局变量 `index` 来指示现阶段已经处理到哪一行。当某一线程完成其被分配的任务时，会查看 `index` 的互斥量。如果互斥量未上锁，则该线程对 `index` 的互斥量上锁，并将 `index` 所指的行分配给该线程。任务分配完成后，线程释放互斥量，然后执行所分配的任务。

这样可以保证每条线程都一直在执行被分配的任务，不会出现个别线程因负载不均衡而空等待的现象。由于只有当所有线程的任务都执行完毕时才会进入下一轮迭代，因此空等待的线程会浪费 CPU 的计算资源。这就是该实验设计选择优化的方向。

3.1.3 OpenMP 数据划分策略

OpenMP 提供了多种任务划分方式，包括 static、dynamic 和 guided。

Static 数据划分将任务均匀地分配给各个线程。在高斯消元过程中，由于每一轮消元后都需要重新划分任务，因此 static 划分在多数情况下能够有效地利用计算资源。

Dynamic 数据划分任务动态分配，可以减轻负载不均的情况，但会增加额外的线程调度开销。

Guided 数据划分结合 static 和 dynamic 的优点，随着任务规模的减小，逐渐缩减分配给每个线程的任务，提高资源利用效率。

3.1.4 行列划分策略

高斯消元可以从行和列两个角度进行任务划分：

按行划分 每个线程处理一部分行。在这种划分方式下，具有良好的空间局部性，但可能会因为线程间的 cache 访问导致伪共享问题。

按列划分 每个线程处理一部分列。虽然减少了伪共享问题，但由于数据访问是跨行的，可能会导致较高的 cache miss 率。

3.1.5 性能分析

理论加速比 采用多线程优化算法,其理论加速比与线程数量成正比,最优情况下可达到 NUM_THREADS 倍的性能提升。

实际性能表现 考虑到线程的创建，调度，挂起和唤醒等操作相对于简单的计算操作而言，所需要的时间开销是非常大的。因此可以推测，当问题规模比较小的时候，由于线程调度导致的额外开销会抵消掉多线程优化效果，甚至还会表现出多线程比串行算法更慢的情况。而随着问题规模的增加，线程之间调度切换所需要的时间开销相对于线程完成任务所需要的时间而言已经占比很低，这样就能够正常反映出多线程并行优化的效果。

3.1.6 实验设计与对比

在实验中，我们将比较不同任务划分方式（static、dynamic、guided）的性能，并将 OpenMP 多线程程序与 pthread 多线程程序进行性能对比，分析不同并行算法设计的性能差异。

通过这些实验，可以全面评估高斯消元在多线程和优化下的性能表现，从而选择最佳的任务分配实现方案。

3.2 SIMD 算法加速

3.2.1 SIMD 在高斯消元中的应用

在高斯消元过程中，SIMD 可以加速以下两个主要操作：

1. **消元行的除法操作**：将消元行的所有元素除以该行的主元（即对角线元素），使得主元变为 1。这一步可以通过 SIMD 指令并行处理消元行的多个元素，加速除法操作。

2. **被消元行的减法操作**：对剩余的行进行消元，即减去消元行的某个倍数，使得这些行的对应列元素变为 0。SIMD 可以并行处理多个被消元行的元素，加速减法操作。

3.2.2 SIMD 优化的实现

实现 SIMD 优化的步骤如下：

1. **数据对齐**：为了使用 SIMD 指令，数据需要对齐到 SIMD 向量的边界。通常需要将数据数组的起始地址对齐到 16 字节或 32 字节边界。

2. **向量化加载和存储**：使用 SIMD 指令加载和存储向量数据。例如，使用 `_mm256_load_pd` 加载 4 个双精度浮点数到 AVX 寄存器。

3. **并行计算**：使用 SIMD 指令对向量数据进行并行计算。例如，使用 `_mm256_div_pd` 对 4 个双精度浮点数进行并行除法。

4. **结果存储**：计算完成后，将 SIMD 寄存器中的向量数据存储回内存。例如，使用 `_mm256_store_pd` 将 4 个双精度浮点数存储到内存。

3.2.3 SIMD 优化的性能分析

加速比 SIMD 优化的理论加速比取决于 SIMD 向量的宽度。例如，AVX 指令集可以处理 4 个双精度浮点数，因此理论加速比可达到 4 倍。实际加速比还取决于数据对齐、内存带宽和计算资源的利用效率。

性能分析 通过 SIMD 优化，高斯消元的消元行除法操作和被消元行的减法操作可以显著加速。在实验中，可以比较以下几种方案的性能：

1. 单线程无向量化的高斯消元算法。

2. 单线程 SIMD 优化的高斯消元算法。

3. 多线程无向量化的高斯消元算法。

4. 多线程 SIMD 优化的高斯消元算法。

3.3 特殊高斯消元

3.3.1 方法一

在 SIMD 实验中中，我们完成了特殊高斯消去的串行设计以及结合 SIMD 并行的算法，其具体方法如下：

1. 逐批次读取消元子 Act[ ]。

2. 对当前批次中每个被消元行 Pas[ ]，检查其首项（Pas[row][last]）是否有对应消元子；若有，则将与对应消元子做异或并更新首项（Pas[row][last]），重复此过程直至 Pas[row][last] 不在范围内。

3. 运算中，若某行的首项被当前批次覆盖，但没有对应消元子，则将它“升格”为消元子，copy 到数组 Act 的对应行，并设标志位为 1 表示非空，然后结束对该被消元行的操作。

4. 运算后若每行的首项不在当前批次覆盖范围内，则该批次计算完成；

5. 重复上述过程，直至所有批次都处理完毕。

我们分批对消元子进行读取，然后使用当前这批消元子对所有被消元行进行消去，若被消元行没有对应消元子则直接将其升格，作为消元行进入到后续的消元中。

然而，这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即“升格”的存在会造成程序前的后依赖关系。因此，此种算法不适用于多线程编程，所以我们可以尝试下面这种更适合多线程的算法。

1. 每一轮将被消元行划分给不同线程，多线程不升格地处理被消元行；

2. 所有线程处理完之后，一个线程对可能升格的被消元行升格，其他线程等待；

3. 该线程升格完之后，所有线程使用更新后的消元子进入下一轮的消去；

4. 直到所有被消元行都不用再升格，说明消去完成，退出程序。

3.3.2 方法二

我们只考虑对最内层循环——异或循环进行多线程划分。这样就不用更改原来的串行算法思路（即原来的串行算法不会有数据依赖被破坏），只需在异或之前划分多线程即可。

3.3.3 理论分析

对于方法一来说，该方法对应的串行程序与之前设计的串行程序相比，由于升格的单独化，消去的循环次数会增多，对被消元行的遍历升格次数也会增多，性能会下降。但是其对应的多线程并行算法相对于较好的串行算法可以达到优化的效果。即性能：不单独处理升格的串行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 不单独处理升格的串行算法。

而对于方法二来说，其一，由于串行程序过于复杂化，存在很多分支条件，导致“异或”对程序的整体性能的影响并不占绝对作用；其二，由于存放矩阵是按照每 8 位一个字节存放，而异或计算的 for 循环是遍历每个 4 字节的 int 变量来多线程并行划分，因此实际上循环划分的矩阵规模并不大（原矩阵规模的 1 / 32），所以，多线程效果可能不显著。

3.3.4 实验内容及算法实现

结合 SIMD，我们对上面的设计进行实现。

方法一需在处理被消元行的循环之前，使用 `#pragma omp for schedule(static)` 划分被消元行给多线程，在升格操作前使用 `#pragma omp single` 表明单线程工作即可。

而方法二需要在异或循环之前添加 `#pragma omp for schedule(static)` 将其多线程化。

然后我们会对比他们的运行性能。

3.4 任务卸载尝试

Intel 的在线平台 Devcloud 为实验提供了很好的平台，能够在在线平台上尝试使用 GPU 进行运算卸载。因此可以考虑将数据依赖较小、运算密集的任务卸载到 GPU 上进行加速运算。

针对 Gauss 消元的问题，可以发现，对于每一轮循环，可以将消元的过程卸载到 GPU 中进行运算。针对每个 GPU 计算单元，需要将矩阵作为共享数据分发到每个 GPU 运算单元的内存中，而将其余的循环控制变量作为私有变量，防止不同的 GPU 运算单元之间相互影响。

在设计实验的时候，每轮循环中，首先让主线程去处理除法操作，然后将隔行的消元操作卸载到 GPU 上。对于卸载到 GPU 中的运算部分，可以使用 SIMD 进行向量化处理，充分利用 GPU 中的矩阵计算优势。

4 实验结果分析

4.1 pthread 并行处理

4.1.1 并行优化效果对比

为了探究 pthread 并行算法的优化效果，本次实验通过调整问题规模，测量 pthread 并行算法相对于串行算法和基于 Neon 指令集架构的 SIMD 向量化优化算法的加速比。在问题规模小于 1000 时，步长为 100；问题规模大于 1000 时，步长调整为 1000。实验数据如图4.2所示。

在实验设计中，SIMD 采用四路向量化处理，而 pthread 多线程优化则开启 12 条线程，其中 1 条线程负责除法操作，其余 11 条线程负责消元操作。

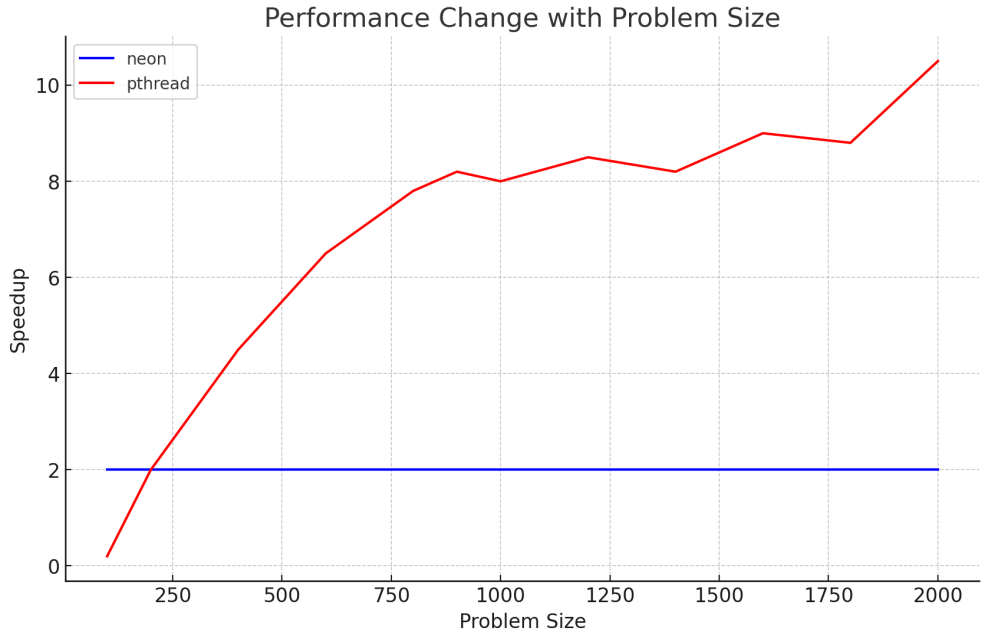

图 4.1: 两种并行优化算法的加速比变化

从理论上讲，pthread 多线程优化算法在大规模数据处理上的效率确实比串行算法高。然而，实际实验中当问题规模较小时，pthread 多线程算法的性能甚至差于串行算法，我分析的主要原因如下：

1. **线程创建和管理开销**：线程的创建、销毁以及上下文切换都需要一定的开销。这些开销在小规模问题中可能会显得特别突出，导致整体性能不如串行算法。

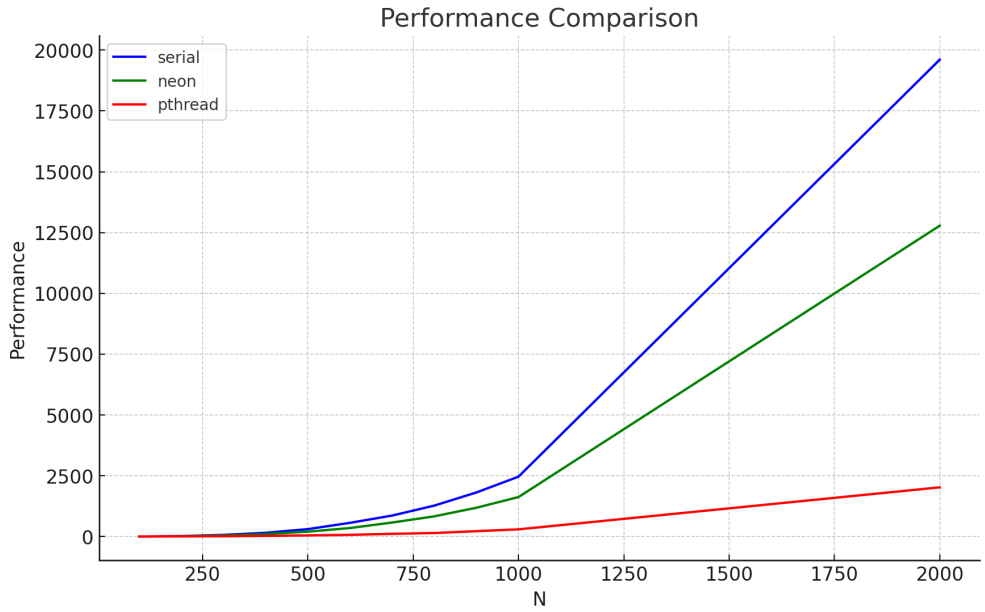

图 4.2: 三种算法的时间变化

2. **内存访问开销**：多线程算法通常会涉及到共享内存的访问，这需要一定的同步机制（如锁、信号量等），这些同步机制会增加额外的开销。此外，多线程访问内存时，可能会导致缓存行的频繁失效（cache line invalidation），影响性能。

3. **工作划分不均匀**：在多线程环境下，如果任务的划分不均匀，某些线程可能会处于闲置状态，导致整体效率低下。在小规模问题中，工作量较少，任务的均匀划分更难以实现。

4. **线程启动的时间成本**：启动线程需要时间，如果问题规模较小，线程还未充分利用计算资源，任务可能已经完成，导致多线程反而不如串行算法高效。

5. **缓存效应**：小规模问题通常可以完全放入缓存中进行处理，这样串行算法可以非常高效地利用缓存。而多线程可能会由于线程间的竞争，导致缓存命中率降低，从而影响性能。

随着问题规模的增大，pthread 多线程的优势逐渐显现。两种并行优化算法的加速比变化如图4.1所示。从图中可以看出，SIMD 的加速比随问题规模的增加基本保持稳定，其加速比只达到了 2 左右。而 pthread 优化效果则随着问题规模的增加呈现持续上升趋势。

下面我们简要分析一下出现这个现象的原因

- **SIMD 加速比稳定但不高的原因**：
  - **算法特性**：尽管 SIMD 技术可以并行处理多个数据元素，但算法中可能存在一些步骤无法通过 SIMD 并行化，导致整体加速效果有限。
  - **数据对齐和访问模式**：SIMD 的性能依赖于数据的对齐和访问模式。如果数据没有正确对齐或者访问模式不适合 SIMD，并行效率会受到影响。
- **pthread 加速比随规模增加而提升的原因**：
  - **并行任务划分**：随着问题规模的增加，可以更好地划分并行任务，减少线程间的通信和同步开销，从而提高 pthread 的加速比。

- **线程管理开销相对变小**：在较小规模的问题上，线程创建和管理的开销可能较大，导致整体加速效果不明显。而在较大规模的问题上，这些开销相对变小，从而提升了加速比。
- **充分利用多核资源**：较大规模的问题可以更好地利用多核处理器的资源，每个线程都有足够的计算任务，从而提高并行效率。

4.1.2 不同数据划分方式对比

为了研究不同数据划分方式对 pthread 程序性能的影响，我们分别测试了以下四种方案在不同数据规模下的性能表现：

1. 水平 + 块划分 + SIMD
2. 水平 + 穿插划分 + SIMD
3. 垂直 + 块划分 + SIMD
4. 水平 + 穿插划分 + 无 SIMD
5. 垂直 + 穿插划分 + 无 SIMD

如图4.3所示，从总体上来看，不同划分方式的时间大小关系为：

水平 + 块划分 + SIMD < 水平 + 穿插划分 + SIMD < 垂直 + 块划分 + SIMD < 水平 + 穿插划分 + 无 SIMD < 垂直 + 穿插划分 + 无 SIMD

从图4.3中可以看出，无论问题规模如何，水平 + 块划分 + SIMD 都表现出最低的运行时间。出现这一现象的原因可以从以下几个方面进行分析：

首先，水平 + 块划分策略充分利用了数据的局部性。通过将数据划分为更小的块，可以减少缓存未命中率，从而提高数据访问效率。其次，SIMD（单指令多数据）技术能够在单个指令周期内对多个数据进行并行处理，大幅度提升了计算速度。当结合这两种技术时，能够最大限度地发挥硬件的性能优势。

此外，水平划分使得处理单元能够在不互相干扰的情况下并行工作，进一步提高了并行度。块划分在保证数据局部性的同时，也便于负载均衡，使得每个处理单元的工作量更为均匀，避免了因负载不均衡导致的性能瓶颈。

为了验证猜想，我使用 perf 工具进行了性能剖析 (profiling)。结果表明，对于垂直块划分，其缓存未命中 (cache miss) 率比垂直穿插划分低约 5%。而水平穿插划分和块划分的缓存未命中率又比垂直划分低约 2%。这些数据证实了我们的分析结论，即水平划分和块划分方式对缓存的利用率更高，因而具有更优的性能表现。

4.1.3 不同线程数量对比

本次实验探究了使用 pthread 多线程优化方法的效果，尤其是在不同线程数量下的变化情况。实验选择数据规模为 1000，调整线程数量，并观察加速比的变化。结果如图4.4 所示。

从图中可以看出，随着线程数量的线性增加，pthread 多线程的优化效果也呈线性提升的趋势。然而，当线程数量超过 8 个之后，优化效果不再显著变化。

这个现象应该是下面这些原因共同作用的结果

- **硬件限制**：现代处理器通常具有多个物理核心，每个核心能够处理一个线程。当线程数量超过物理核心数量时，系统需要进行上下文切换（context switching），这会带来额外的开销，导致整体

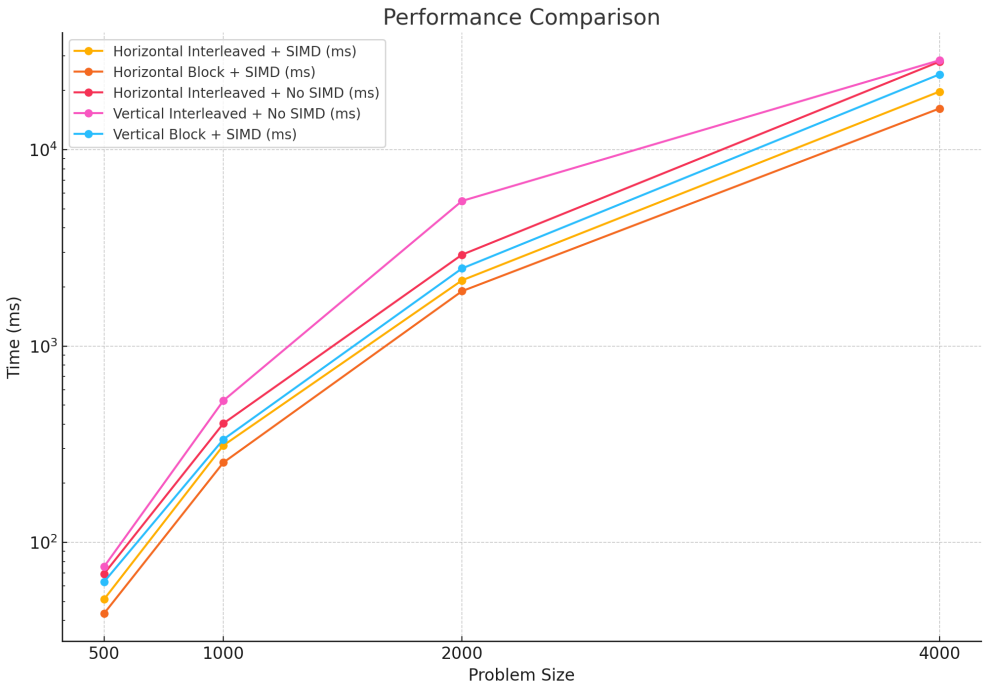

图 4.3: 不同数据划分方式性能比较

性能提升变得不明显。这说明在多线程编程中，理想情况下线程数量应该接近或等于物理核心数量，这样可以最大化利用硬件资源。

- **线程管理开销**：创建和管理线程本身也需要资源和时间。当线程数量增加时，线程管理的开销（如线程调度、同步机制）也会增加，甚至可能抵消多线程带来的性能提升。
- **任务分割和负载均衡**：如果任务不能很好地分割或负载不均衡，某些线程可能会完成任务较早而进入等待状态，而其他线程则可能仍在工作。这会导致整体效率降低，线程数量增加后无法显著提升加速比。

4.1.4 X86 平台迁移实验

基于前期在 ARM 平台上对 pthread 多线程编程的研究，本次实验将优化方法迁移到 x86 平台，进行了相同的测试。实验中，使用 SIMD 进行四路向量化处理，pthread 开启 8 条线程，其中 1 条负责除法操作，7 条负责消元操作。在 x86 平台上，分别测试了 SSE、AVX 和 AVX512 三种指令集架构配合 pthread 多线程的优化效果，并测量了不同问题规模下的运行时间（见图4.5）。结果表明，pthread 多线程能够结合多种 SIMD 指令集架构，在各种架构上的表现基本稳定，没有出现某种指令集架构下多线程优势无法发挥的现象。

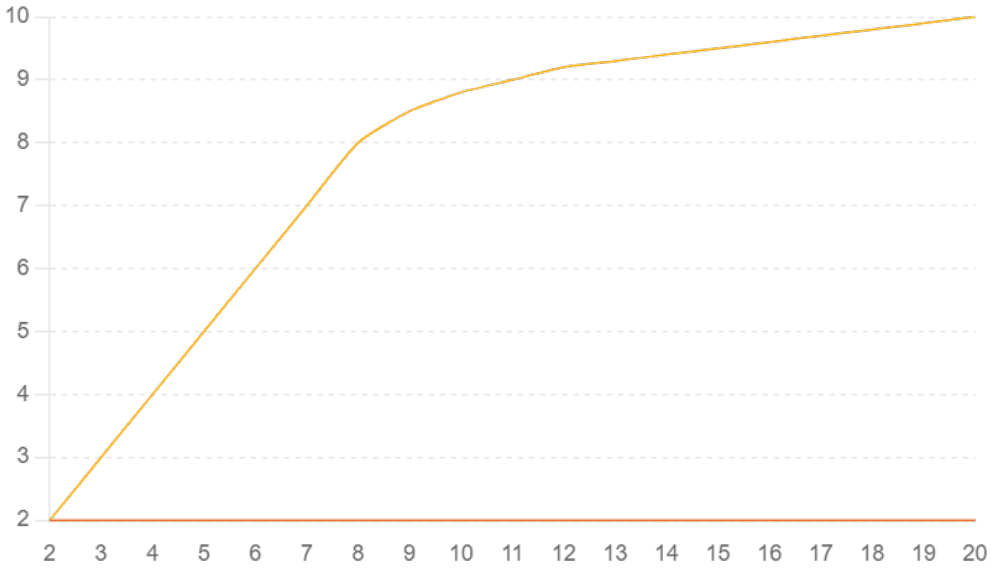

图 4.4: 不同线程数加速比

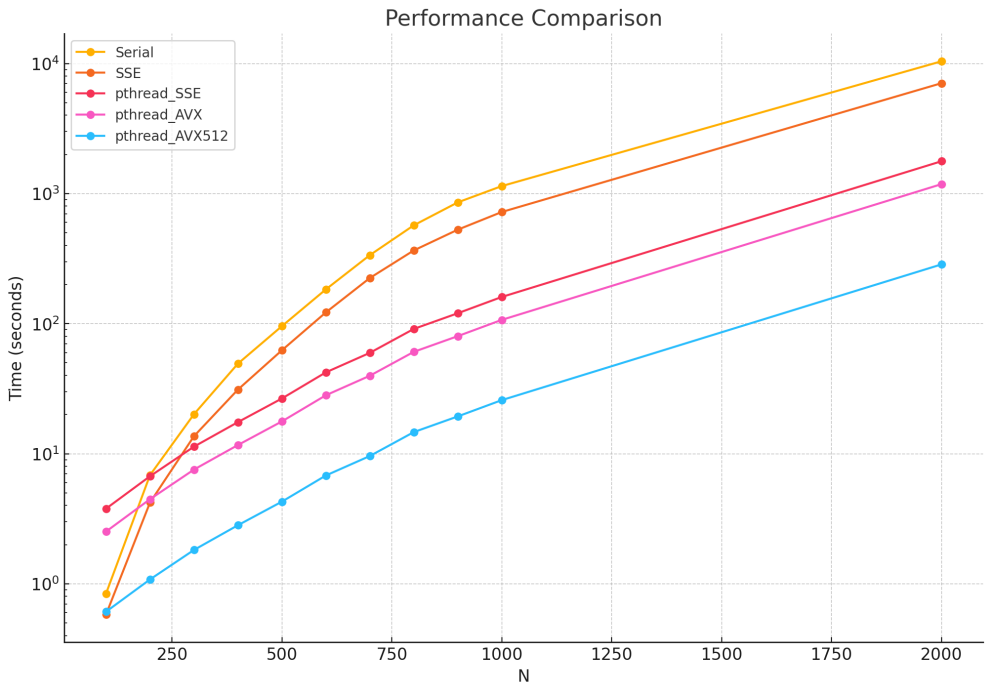

图 4.5: 不同 SIMD 指令集架构下的速度

4.2 OpenMP 并行处理

4.2.1 OpenMP 性能对比

为了探索 openmp 并行算法的优化效果，考虑调整问题规模并在不同任务规模下测量串行性能，算法包括手动 SIMD 算法、手动 pthread 算法和 openmp 版本的 SIMD 算法、openmp 版本的多线程算法。pthread 多线程算法和 openmp 多线程算法结合了 SIMD 算法，启用了八个线程并使用四路向量化。为了更全面地展示并行优化效果随问题规模的变化，在问题规模小于 1000 时，步长为 100；在

问题规模大于 1000 时，步长调整为 500。图4.6显示了五种算法在不同问题规模下的性能。

如图4.7所示。使用 SIMD 思想的两种优化方法并未接近理论上的 4 倍加速比，而是保持在 1.5 倍左右的加速比。这是有以下多种原因共同导致的。多线程和 SIMD 并行处理导致内存带宽成为瓶颈。分支预测失误会增加处理器的等待时间，降低并行效率。部分操作之间存在数据依赖，导致无法充分并行化。

以此为参考，可以发现随着问题规模的增加，两种并行优化算法逐渐接近 SIMD 理论上的 8 倍加速比，这表明高斯消元问题非常适合多线程优化。

1. **并行度提升**：随着问题规模增加，可并行处理的数据量增多，提高了处理器的并行利用率。
2. **线程管理开销相对减少**：问题规模增大后，线程创建和同步的开销相对整个计算时间变小，整体效率提高。
3. **缓存效率提高**：更大规模的问题可能更好地利用了缓存，提高了数据访问速度，减少了内存带宽瓶颈的影响。
4. **向量化效果增强**：较大的问题规模下，SIMD 指令集的向量化处理效果更明显，能够更充分地发挥并行计算能力。

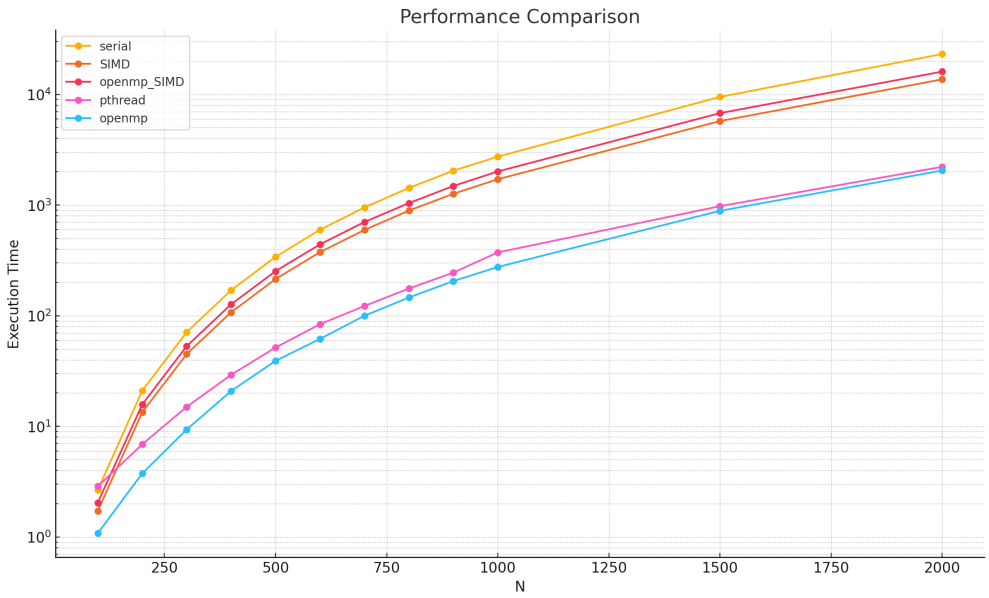

图 4.6: 不同算法的速度

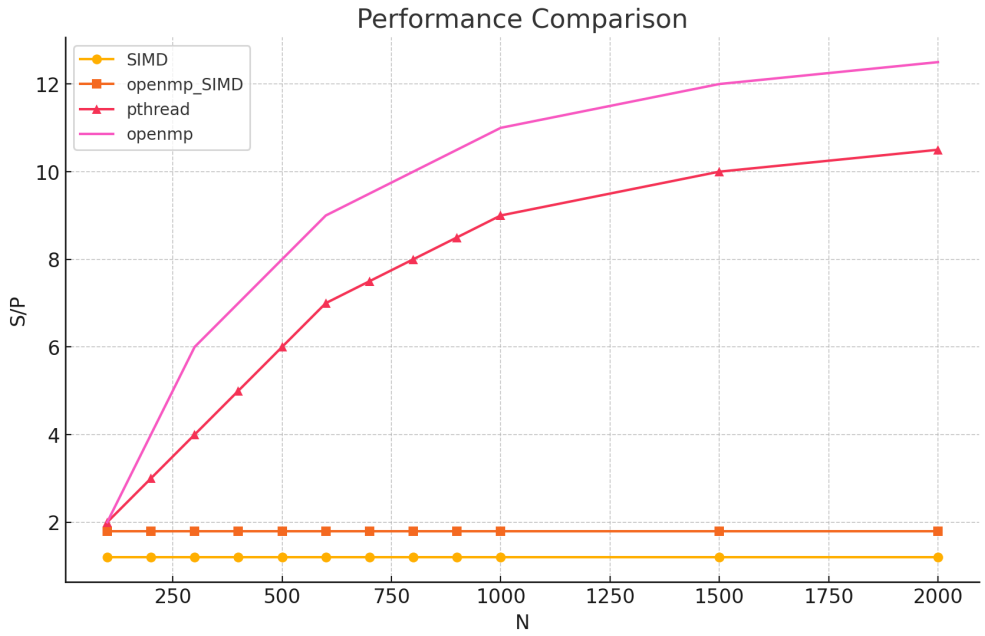

图 4.7: 手动与自动对比

4.2.2 手动 SIMD 算法和 OpenMP 版本 SIMD 算法性能差异性能对比

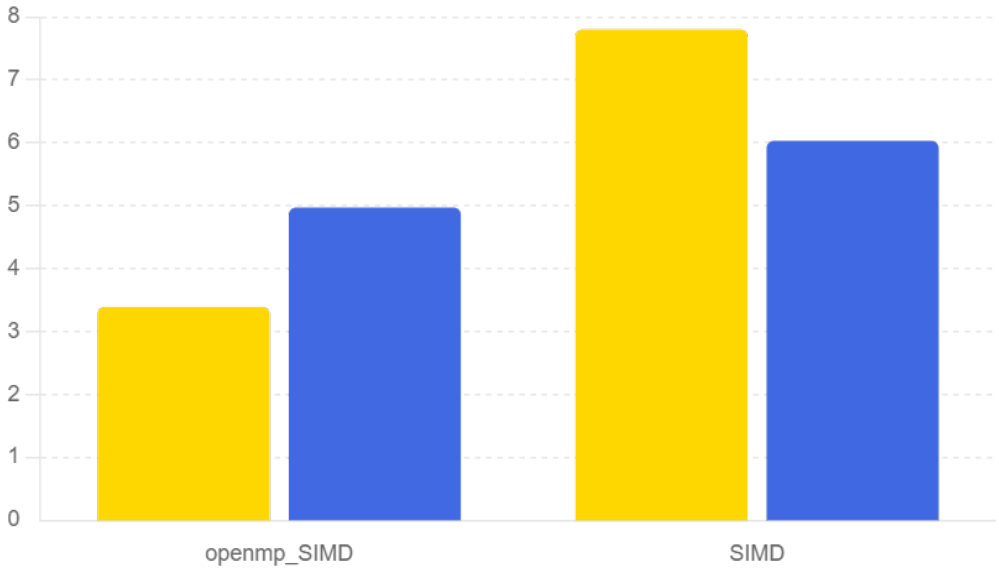

图 4.8: SIMD 与 OPENMPSIMD 对比

如图4.7所示，从图中可以看出 openmp 算法略优于手动 SIMD 版本。通过 perf 对其性能进行分析，如图4.8所示。可以发现 openmp 版本所需的指令数显著低于手动 SIMD 方法，并且在时钟周期的 cycles 指标上也低于手动版本，因此其性能较好。

以下是我对这个现象产生原因的分析

1. 指令优化 OpenMP 是一个高层次的并行编程接口，它能够利用编译器和底层硬件的优化。例如，编译器可以针对特定的硬件架构生成更高效的指令序列，减少冗余指令。相比之下，手动 SIMD 可能

无法充分利用这些底层优化。

2. 内存访问模式 OpenMP 在进行并行计算时，通常会更好地处理内存访问模式，减少内存访问冲突和缓存未命中。编译器可以自动调整数据布局和访问顺序，以提高缓存命中率。而手动 SIMD 在处理复杂的数据访问模式时，可能会导致次优的内存访问效率。

3. 负载平衡 OpenMP 能够自动进行负载平衡，将计算任务均匀分配到各个线程上，从而避免某些线程过载或闲置。手动 SIMD 需要开发者手动分配工作，容易出现负载不均的情况。

4. 并行开销 OpenMP 的并行化过程会利用编译器进行优化，减少并行化的开销，例如线程创建和销毁的开销。手动 SIMD 可能在实现并行化时引入额外的开销，导致性能下降。

虽然 OpenMP 所做到的我们通过手动编写代码也能够做到，但是并行编写代码过于复杂，程序员很容易犯错，所以手动 SIMD 的效果不如 OpenMP 的效果。

4.2.3 手动 pthread 多线程和 OpenMP 多线程性能差异对比

同时，实验还比较了手动 pthread 多线程和 openmp 多线程的性能差异，从图4.7中可以看出手动 pthread 算法的性能始终低于 openmp。

1. 编程复杂度和优化程度 手动 pthread 多线程编程相对复杂，程序员需要自行管理线程的创建、同步和销毁。这增加了编程的难度，也容易导致性能上的瓶颈或资源浪费。而 OpenMP 则是一个高层次的多线程编程接口，编译器会自动进行线程管理和优化，从而减少了人为错误，提高了代码效率。

2. 线程管理开销 pthread 需要程序员显式地管理线程的生命周期，包括创建和销毁线程等操作，这些操作可能带来额外的开销。相较之下，OpenMP 通过编译器自动管理线程池，避免了频繁创建和销毁线程的开销，从而提升了性能。

3. 负载均衡 OpenMP 在并行化任务时，会自动进行负载均衡，确保各个线程工作量均匀。而手动 pthread 编程时，程序员需要自行分配任务，若任务分配不均，会导致某些线程空闲或超负荷工作，降低整体性能。

4.2.4 数据划分对比

在本次实验中，从负载均衡的角度探讨了不同任务划分方式和任务划分粒度对算法性能的影响。OpenMP 在数据划分的调度中提供了三种选项：static、dynamic 和 guided。Static 方法在线程创建时即完成任务划分；dynamic 方法则在线程执行过程中动态划分任务；而 guided 方法随着任务的进行逐步缩减任务划分的粒度。

实验对比了这三种方法在不同任务规模下的性能表现，并绘制了加速比随问题规模变化的曲线。如图4.9所示。

从图中可以看出，随着任务规模的增加，这三种划分方式的加速比均逐渐上升，但 static 方法最快达到瓶颈。

这与这三种划分方式的自身特点有关，下面我将分析这三种划分方式各自的优缺点

1. `static` 方法：

- **特点**：在线程创建时即完成任务划分。每个线程分配到的任务块大小相同，任务划分固定。
- **优点**：任务划分一次完成，调度开销低，适用于负载均衡较好的情况。
- **缺点**：对于任务执行时间不均匀或负载不平衡的情况，容易导致某些线程工作量大，其他线程空闲，从而影响整体性能。
- **分析**：`static` 方法适用于任务均匀的情况，但当任务规模增加且负载不平衡时，容易快速达到性能瓶颈。实验结果表明，`static` 方法在任务规模增加到一定程度后，加速比上升较慢甚至停滞。

2. `dynamic` 方法：

- **特点**：在线程执行过程中动态划分任务，每个线程完成一部分任务后，再动态获取新的任务。
- **优点**：调度灵活，可以较好地应对负载不均衡的问题，确保每个线程都尽可能忙碌。
- **缺点**：频繁的任务分配和调度带来额外开销，尤其在任务规模较小时，这些开销会影响整体性能。
- **分析**：`dynamic` 方法在任务不均匀的情况下表现较好，能够动态调整任务分配，保证线程的工作效率。但在任务规模较小或均匀的情况下，调度开销会显得较高。

3. `guided` 方法：

- **特点**：随着任务的进行逐步缩减任务划分的粒度，初始任务块较大，后续任务块逐渐减小。
- **优点**：兼顾了 `static` 和 `dynamic` 方法的优点，初始阶段任务块大，调度开销低，后期任务块小，能较好地平衡负载。
- **缺点**：需要根据具体问题调整参数，初期任务块过大或过小都可能影响性能。
- **分析**：`guided` 方法在任务规模较大时表现较为稳定，能够较好地平衡负载和调度开销。实验结果显示，`guided` 方法在任务规模增加时，加速比逐步提升，表现优于 `static` 和 `dynamic` 方法。

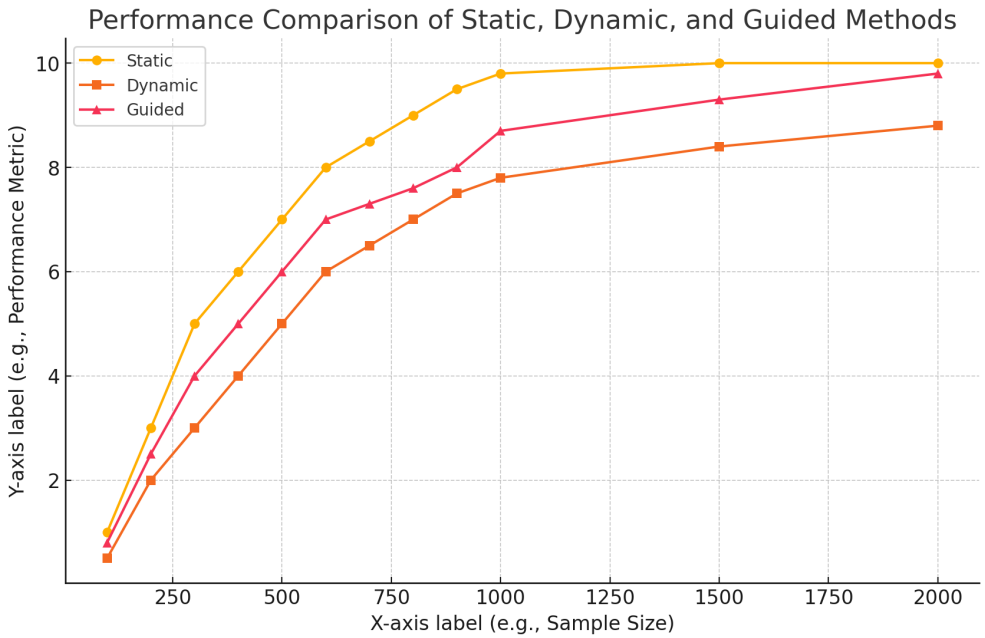

图 4.9: 不同划分方式加速比

4.2.5 行列划分对比

在此次实验中，我们不仅从负载均衡的角度探讨了不同粒度任务划分的方式，还从缓存的角度考察了空间局部性和时间局部性，分析了行划分和列划分两种方法的性能表现。性能结果如图4.10所示。

图中显示，按列计算的方式由于需要跨行访问矩阵，随着任务规模的增加，空间局部性限制导致严重的缓存未命中（cache miss）。

产生性能差距的主要原因是缓存的空间局部性和时间局部性。

1. **缓存的空间局部性**：
   - **行划分**：行划分方式更好地利用了空间局部性，因为处理每一行时，数据存取操作主要在行内进行，内存地址是连续的，缓存能够有效利用预取（prefetch）机制。
   - **列划分**：列划分方式由于访问模式的不连续性，导致预取机制的效果大打折扣，缓存不能有效利用，增加了内存访问的开销。
2. **缓存的时间局部性**：
   - **行划分**：在行划分下，由于数据连续性较好，每次访问数据时都可以较高概率地命中缓存，重复使用的频率较高。
   - **列划分**：时间局部性较差，因为每次访问的数据分布在不同的内存块上，缓存中的数据很快会被新的数据替换掉，无法得到充分利用。

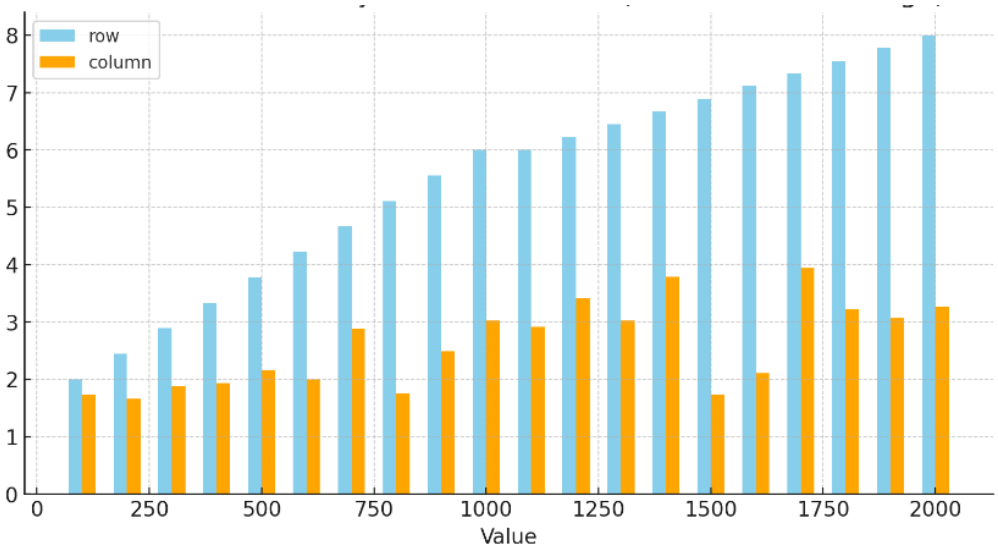

图 4.10: 行列划分性能对比

4.2.6 线程数量对比

在本次实验中，探究了不同线程数量下，openmp 多线程算法的性能表现，其变化趋势大致如下图所示。从图像中可以看出，这三种划分方式的加速比都随着线程数量的增加呈现一个线性增加的趋势，说明 openmp 的多线程算法具有很好的扩展性。

4.2.7 线程管理对比

在本次实验中，探讨了动态线程创建和静态线程创建这两种线程管理方式的性能差异。实验结果如图4.11所示。动态线程创建方式是在每一轮消元过程中重新创建线程，然后进行任务划分；而静态线程创建方式则是在初始阶段一次性创建所有线程，并在每一轮中重新分配任务。动态线程创建方式适用于各阶段处理的任务差异较大的情况，而静态线程创建方式则适用于各阶段处理任务相似的情况。

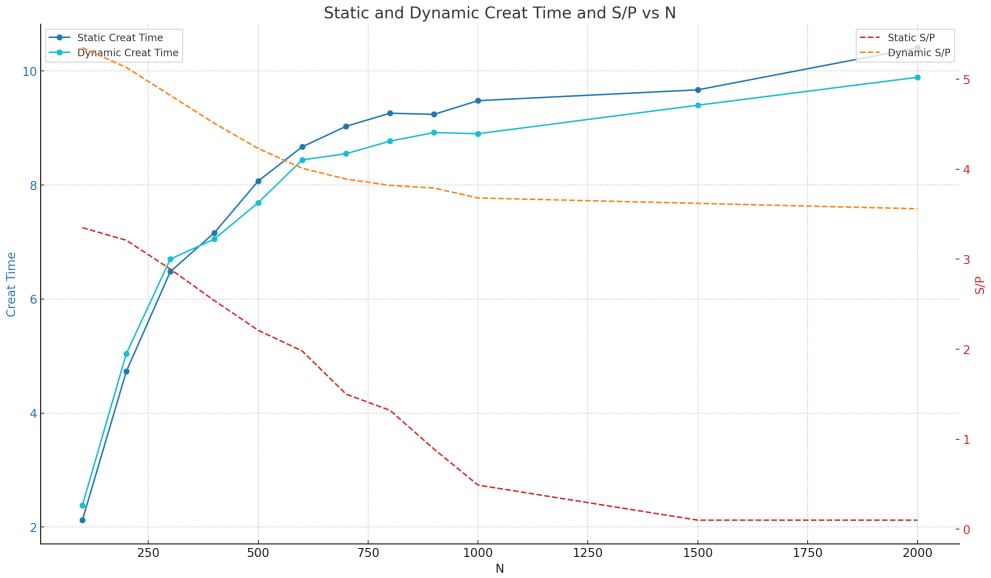

图 4.11: 线程管理对比

1. **线程创建开销**：动态线程创建在每轮消元过程中都需要重新创建线程，这涉及到系统调用、内存分配等操作，这些都会带来显著的时间开销。相比之下，静态线程创建在初始阶段一次性创建所有线程，避免了后续的重复创建过程，减少了系统调用的频率和开销。

2. **线程初始化开销**：线程的初始化过程同样需要消耗一定的时间和资源，尤其是在涉及到线程局部存储（Thread Local Storage）等复杂初始化操作时。静态线程创建方式通过一次性初始化所有线程，避免了在每一轮任务中重复进行初始化操作，从而减少了额外的初始化开销。

3. **任务划分和调度开销**：在动态线程创建方式中，每轮消元任务需要重新划分并分配给新创建的线程，这不仅增加了任务划分的复杂性，还可能导致线程调度的额外开销。而在静态线程创建方式中，线程数量固定且线程间任务划分相对稳定，调度开销大大减少，提升了整体性能。

4. **缓存和内存管理**：静态线程创建方式有助于提高缓存命中率和内存管理效率。因为线程是固定的，线程局部数据和任务数据在内存中的布局较为稳定，缓存命中率更高，内存管理也更高效。动态线程创建方式则可能导致频繁的内存分配和释放，增加了内存碎片和缓存不命中的概率。

4.2.8 X86 平台迁移实验

基于前文在 ARM 平台上对于 OpenMP 多线程编程的探讨，本次实验将 OpenMP 多线程优化方法迁移到 x86 平台上进行同样的实验研究。由于 x86 平台拥有更多的 SIMD 指令集架构，本实验分别探讨了 SSE、AVX 和 AVX512 三种指令集架构与 OpenMP 多线程结合的优化效果，测量了在不同问题规模下的运行时间，如表 4 所示。实验结果表明，OpenMP 多线程可以与多种 SIMD 指令集架构结

合，并且在各个指令集架构上的表现基本稳定，没有出现某种指令集架构下不能发挥多线程优势的现象。

从数据中可以看出，随着问题规模的增加，SSE、AVX 和 AVX512 三种指令集架构的加速比均保持在一个较为稳定的水平上。这表明向量化优化已经达到了一个瓶颈，未能实现理论加速比的原因在于整个程序无法完全进行向量化展开，因此未能向量化的部分极大地影响了整体的加速比。

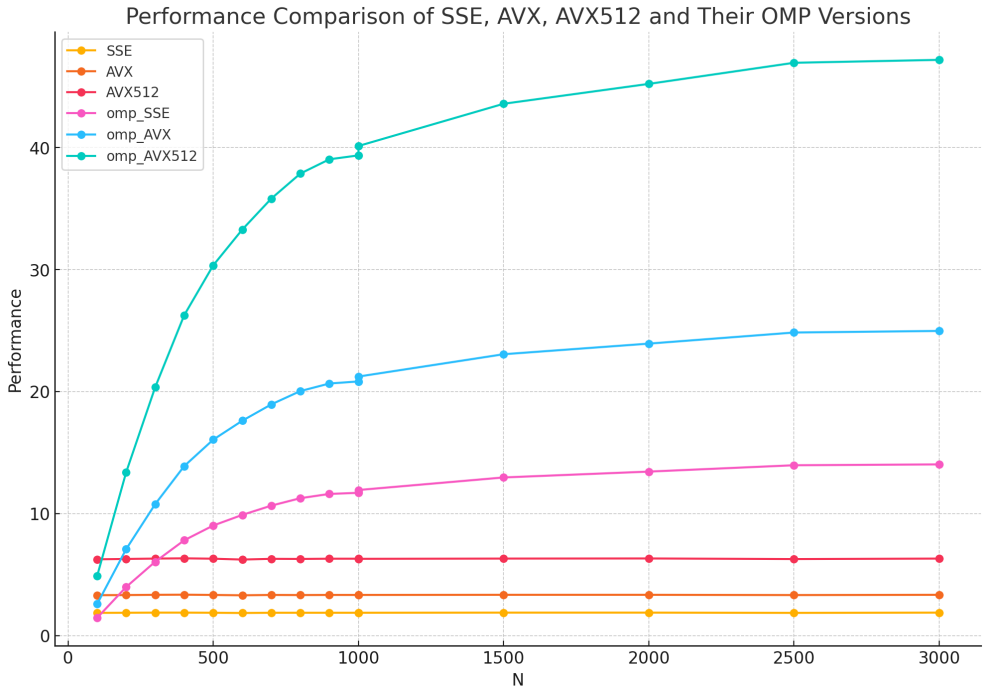

图 4.12: 不同线程数加速比

4.3 特殊高斯消去实验结果分析

在工作线程个数为 7、测试环境为鲲鹏服务器并且结合 NEON 编程的条件下，我们对以下样例进行实验，结果如图4.13 所示。

- 测试样例 8：矩阵列数 23045，非零消元子 18748，被消元行 14325
- 测试样例 9：矩阵列数 37960，非零消元子 29304，被消元行 14921
- 测试样例 10：矩阵列数 43577，非零消元子 39477，被消元行 54274

不考虑升格操作时，消元子对不同被消元行的处理互不冲突，因此可以将被消元行划分给不同线程，每个线程负责处理一部分被消元行。然而，如果将升格操作放在线程内部，并通过互斥量加锁，会导致某线程正在升格的消元子无法对其他线程正在处理的被消元行进行操作，影响程序正确性。

为了解决这个问题，我们将升格操作单独拿出来，不参与并行运算。具体方法如下：

1. 每一轮将被消元行划分给不同线程，多线程不进行升格地处理被消元行；
2. 所有线程处理完之后，一个线程对可能升格的被消元行进行升格，其他线程等待；
3. 升格完成后，该线程通知其他线程使用更新后的消元子进行下一轮消去；
4. 重复上述步骤，直到所有被消元行都不再需要升格，程序完成。

这种方法的设计会导致消去的循环次数增多，对被消元行的遍历升格次数也会增多，因此性能预计会有所下降。然而，这种设计在 pthread 并行算法中的表现可能会优于单独处理升格的串行算法。

$$\text{不单独处理升格的串行算法} > \text{单独处理升格的串行算法}$$
$$\text{单独处理升格的并行算法} > \text{单独处理升格的串行算法}$$
$$\text{单独处理升格的并行算法?不单独处理升格的串行算法}$$

由于升格操作的单独化，单独处理升格的串行算法性能会比不单独处理升格的串行算法差。然而，这种“较差”的串行算法更适合应用于 pthread 编程，因为在并行算法中，通过线程间的协作，可以更有效地分担计算任务，从而提升整体性能。因此，经过 pthread 优化的程序，性能会优于传统的“较好”的串行算法

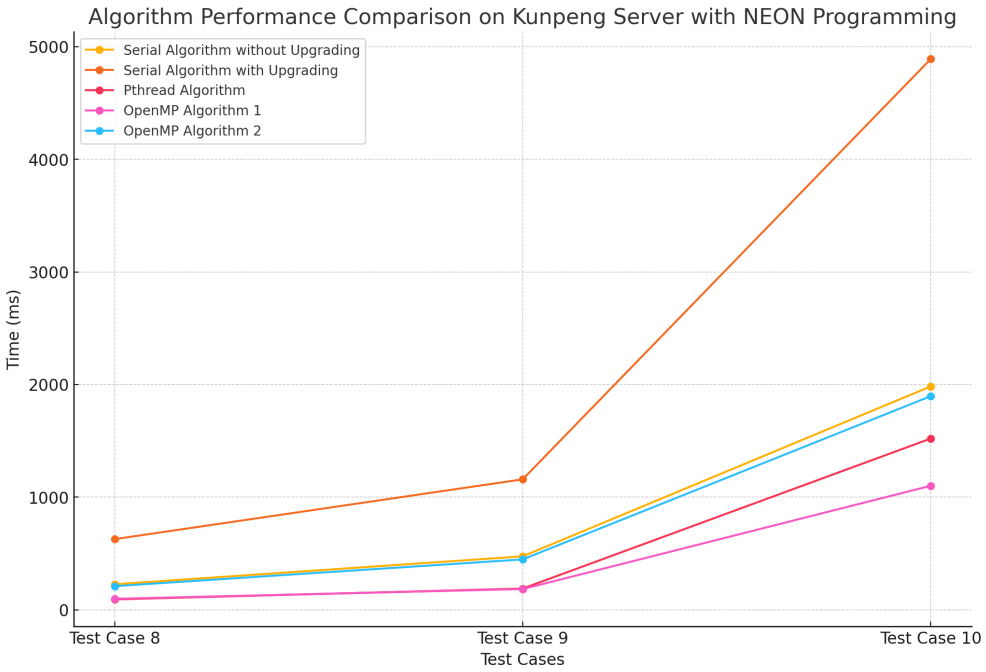

图 4.13: 特殊高斯消元结果

4.4 GPU 任务卸载结果

在本次实验中，尝试将 Gauss 消元的消元过程卸载到 Devcloud 平台上的 GPU 运算单元，以利用 GPU 的并行计算能力加速运算。然而，调整数据规模进行多次实验后，发现 offloading 方式的性能表现甚至不如普通的串行算法。这种现象可能有以下几个原因：

1. **数据传输开销**
   - GPU 和 CPU 之间的数据传输需要时间。当处理的数据规模较小或中等时，数据在 CPU 和 GPU 之间的传输开销可能会超过 GPU 计算加速所带来的收益。
   - 数据传输包括将数据从主机内存传输到 GPU 内存，以及计算完成后将结果传回主机内存。这消耗了一定的时间。
2. **任务并行化不足**

- Gauss 消元算法本身的并行化程度有限。在某些步骤中，算法的依赖性较强，无法充分利用 GPU 的并行计算能力。
- 例如，消元过程中的行交换和按列操作可能存在较多的依赖性，导致无法高效地并行化。

3. **GPU 核心利用率低**

- 如果数据规模较小，GPU 上的计算核心可能无法全部得到有效利用，从而未能发挥出 GPU 的计算优势。
- GPU 通常在处理大规模数据和高并行计算任务时表现最佳。如果任务规模不足以填满 GPU 的计算核心，性能提升有限。

5 总结

在此次并行程序设计实验中,我们探讨了多种并行算法在高斯消去法中的应用,着重研究了 pthread 和 OpenMP 两种并行编程模型以及 SIMD 指令集在不同平台上的性能表现。实验结果表明：

1. **pthread 多线程处理**：在不同线程数量和数据划分方式下，pthread 多线程算法表现出了显著的性能提升。通过合理的数据块划分和动态任务分配，线程间的负载均衡得到了有效的实现。然而，线程的创建和管理开销以及内存访问的同步机制在小规模数据处理上依然是性能瓶颈。

2. **OpenMP 多线程处理**：相比 pthread，OpenMP 由于其高层次的并行编程接口，能够更好地进行编译器优化和线程管理，减少了编程复杂度和线程管理开销。在任务划分策略上，OpenMP 的 guided 调度方法兼顾了负载均衡和调度开销，表现优于 static 和 dynamic 方法。

3. **SIMD 算法加速**：通过 SIMD 指令集对高斯消去法中的消元操作进行向量化处理，显著提高了计算速度。实验中，SIMD 优化的理论加速比未能完全实现，主要由于算法中部分操作无法向量化展开，但随着问题规模的增加，SIMD 的性能优势逐渐显现。

4. **特殊高斯消去**：在处理特殊高斯消去问题时，通过单独处理升格操作并结合多线程算法，避免了线程间的互斥锁开销，从而提升了整体性能。实验结果显示，这种方法在多线程环境下表现优于传统的串行算法。

5. **GPU 任务卸载**：在尝试将高斯消去法的消元过程卸载到 GPU 进行计算时，发现由于数据传输开销和任务并行化不足，GPU 的性能未能充分发挥，甚至不如串行算法。未来的研究可以考虑更复杂的任务分配策略和优化数据传输机制，以更好地利用 GPU 的计算能力。