南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计

期末研究报告——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 7 月 4 日

摘要

本次实验围绕普通高斯消去算法在多种并行计算架构下的优化实现展开,系统探索了 SIMD、Pthread/OpenMP 多线程、MPI 分布式通信与 CUDA GPU 并行编程对高斯消去性能的加速效果。通过构建统一的三段式并行设计框架，实现了跨平台的功能对齐与性能对比。在此基础上，结合不同并行粒度、内存模型与同步机制，对各架构在执行效率、可扩展性和资源利用率方面进行了实验分析与总结。

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

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

实验环境主要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

此外还有北京超算平台，GPU 型号是 **NVIDIA Tesla T4**。

三、 实验设计与并行框架总览

本实验旨在针对高斯消去算法在不同并行计算架构上的性能表现和加速潜力进行全面分析与对比。高斯消元作为一种典型的计算密集型算法，其消去阶段具有显著的数据并行特征，适合在多种并行体系结构上展开加速优化。实验覆盖以下四类主流平台：

- **SIMD 架构**：基于 SSE、AVX 和 ARM NEON 指令集的向量化实现，利用数据并行提升内层循环效率；
- **多核 CPU 架构**：使用 Pthread 和 OpenMP 编程模型在共享内存系统上进行线程级并行；
- **分布式集群架构**：基于 MPI 编程模型在多节点系统上进行进程间通信与协作计算；
- **GPU 架构**：使用 CUDA 在 NVIDIA GPU 上实现海量线程的显存层次并行计算。

为便于统一比较不同并行架构下的优化效果与设计策略，本文对普通高斯消去算法的实现流程进行了标准化抽象，构建出统一的三段式并行框架，适配各类平台的并行模式：

1. **主元标准化阶段（除法）**：将主元行归一化，确保对角线元素为 1；
2. **消去阶段（核心并行部分）**：使用主元行对其下方行进行并行消元，构建上三角矩阵；
3. **回代过程（可选并行）**：根据上三角矩阵，从最后一行反向求解未知数。

每种架构对上述三阶段中并行程度不同的部分进行了差异化优化设计，尤其是在消去阶段展开核心并行。表 1 对不同平台的并行设计特征进行了对比总结：

表 1: 不同并行架构下的高斯消去设计对比

并行架构	并行粒度	同步方式	内存模式	核心优化点
SIMD	向量级操作	无需线程同步	显式对齐/打包	向量指令加速内层矩阵操作
Pthread/OpenMP	行/列/任务级线程	信号量、Barrier	共享内存	动态调度、同步机制优化
MPI	行/列/二维块划分	显式通信	分布式内存	通信开销隐藏与负载均衡
CUDA	线程块、Warp 级	`_syncthreads()`	多级显存	块内线程并行、共享内存协同

本实验从平台、粒度、同步与内存优化等角度系统性探索高斯消去在多架构并行中的性能差异与优化策略。

四、 算法设计分析：架构间的共通与差异

不同架构在统一的三段式并行框架下，通过调整任务划分粒度、并行调度策略、数据访问模式和通信机制等，形成各具特点的优化实现。以下从共通与差异两个角度展开分析。

(一) 共通之处

- **核心关注阶段一致**：各平台均将计算复杂度最高的消去阶段作为并行优化的重点，并在主元标准化阶段采取串行或单线程方案，以确保数据一致性；
- **任务划分以“行/列”为主**：无论是 OpenMP 的 schedule、MPI 的行列划分、CUDA 的 block-thread 网格结构，或 SIMD 向量指令加载，均在任务划分上围绕矩阵的行/列维度展开；
- **回代阶段保留串行**：由于回代阶段存在前后项数据依赖，计算量相对较小，多数平台（包括 MPI、CUDA、OpenMP）选择在主线程/主进程中完成，确保结果准确性；
- **矩阵初始化与秩稳定性保障机制**：在多线程和 GPU 平台上均通过构造上三角矩阵并添加扰动操作，避免出现不可逆或主元过小导致精度问题的矩阵输入。

(二) 主要差异

- **SIMD**：聚焦于内层循环的向量重构。在 AVX/SSE/NEON 平台上，将标量运算替换为 128/256 位寄存器向量运算，实现数据级并行。优化手段包括内存对齐、循环展开、寄存器复用等，但无法并行主元选择与行控制逻辑。
- **Pthread/OpenMP**：以多核 CPU 为基础，强调线程粒度的任务划分和同步机制控制。Pthread 更灵活地实现静态/动态线程控制与信号量同步，OpenMP 借助编译指令简化并行结构构建。特别是在回代保持串行基础上，OpenMP 可与 SIMD 指令集联合使用，实现线程级与指令级双重加速。
- **MPI**：采用消息传递方式处理分布式架构。在算法实现中重点解决通信模式选择（阻塞/非阻塞/单边通信）、任务划分平衡（行块/列块/二维块划分）和流水线优化等问题。MPI 在通信延迟与数据同步方面开销显著，在规模较小时常表现出并行负效益，但结合 OpenMP 和 SIMD 后可显著提升扩展性与可用性。

- **CUDA(GPU)**:通过线程块与线程的二维映射结构,将矩阵的列或行操作映射到 threadIdx 和 blockIdx 上，并利用 _syncthreads() 实现数据同步。线程间通过共享内存加速数据重用，并通过 block 内线程协作实现矩阵块级消元，适用于大规模稠密矩阵并行处理，展现数量级的加速潜力。

小结：虽然各平台的硬件特性与编程模型不同，但在高斯消元这一结构规整、数据密集的计算任务上，均展现出可观的并行优化空间。将统一的三段式算法框架作为比较基准，有助于深入理解不同架构下并行策略的优劣，并为综合优化提供可迁移的设计依据。

新增内容：本报告首次提出 **“统一三段式优化框架”**，在每种架构下分别落地实现，便于横向性能分析与内核一致性检验。

(三) 实验结果分析

图1 和表2、表3 展示了各架构在不同问题规模下的运行时间与加速比，反映出其性能瓶颈与扩展趋势。

表 2: 不同架构在不同问题规模下的运行时间（单位：ms）

问题规模	x86 串行	AVX	SSE	Pthread	OpenMP	MPI	GPU
32	0.04	0.004	0.003	1.083	3.193	0.0456	0.291072
64	0.325	0.018	0.014	1.743	6.01	0.07717	0.597056
128	2.188	0.122	0.094	2.895	11.189	0.21366	3.3792
256	15.568	1.076	0.652	6.261	23.393	1.18161	5.2239
512	140.059	8.516	5.634	18.328	46.415	13.0117	17.1272
1024	1139.062	65.033	46.612	70.068	120.592	80.6078	50.1558
2048	9196.672	1537.806	507.91	1335.012	1500.110	1644.52	172.323
4096	68115.675	16924.36	9312.651	18291.63	19719.157	18095.8	644.931
8192	537561.435	142130.013	84905.539	165252.816	163513.256	161622	2394.45

表 3: 不同架构在不同问题规模下的加速比

问题规模	x86 串行	AVX	SSE	Pthread	OpenMP	MPI	GPU
32	0.04	10	13.3333	0.037	0.0125	0.8772	0.1374
64	0.325	18.0556	23.2143	0.1868	0.054	4.2108	0.5443
128	2.188	17.9344	23.2766	0.7559	0.0196	10.2454	0.6475
256	15.568	14.4643	23.8811	2.4875	0.6669	13.171	2.98
512	140.059	16.4466	24.8673	7.6525	3.0047	10.7614	8.1776
1024	1139.062	17.5153	24.4386	16.2229	9.4582	14.1412	22.71
2048	9196.672	5.9835	4.4701	6.9058	6.1243	5.5925	53.3688
4096	68115.675	3.9973	7.3241	3.71	3.4352	3.6378	105.617
8192	537561.435	3.781	6.3261	3.2537	3.2871	3.2971	224.5

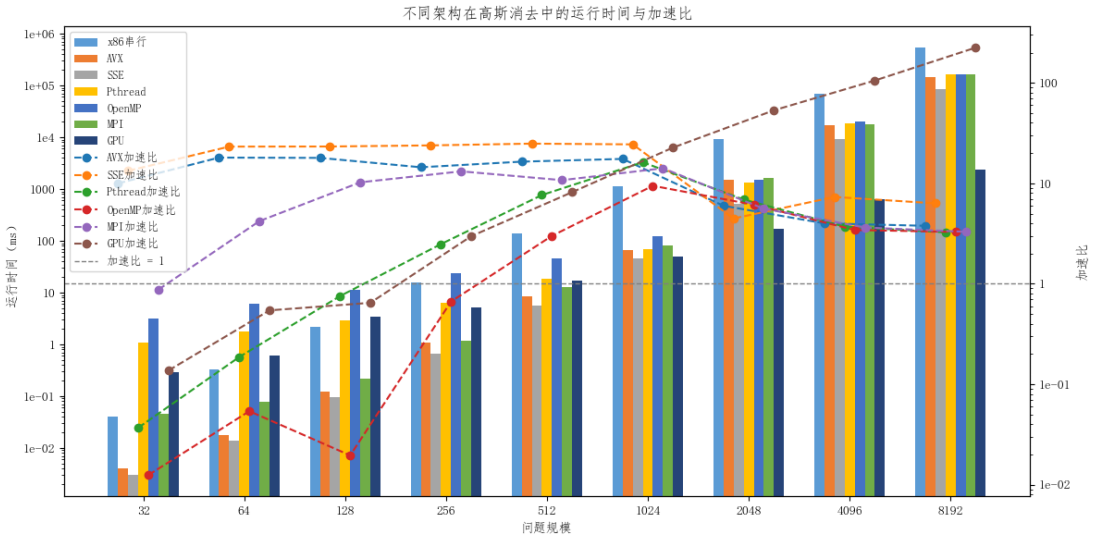

图 1: 不同架构在高斯消去中的运行时间与加速比

1. SIMD 架构（AVX / SSE）： AVX 与 SSE 在小规模矩阵中表现最优，最大加速比分别达到 **18∼24 倍**，主要归因于向量寄存器对内层循环的吞吐提升。然而，在问题规模不断增大后，其性能增益趋于饱和，甚至出现回落，说明向量化优化在计算密集阶段的边际收益有限，并且受限于寄存器数量与缓存容量。

2. Pthread / OpenMP 多线程架构： Pthread 和 OpenMP 在小规模问题下，由于线程的创建、销毁与同步的额外开销，性能明显比串行要差。Pthread 在中等规模（256∼1024）内加速效果逐步提升，最高可达 **16 倍**，表明线程调度与缓存复用较为有效；但在更大规模下，性能不升反降，原因在于线程间共享资源竞争激烈，缓存失效频繁。OpenMP 的加速比整体低于 Pthread，并且在小规模问题下体现出编译指令自动划分对小任务粒度控制不力，唯有与 SIMD 联合优化时才具有优势。

3. MPI 分布式架构： MPI 在规模较小时存在明显通信开销，导致加速效果甚至劣于串行。但随着问题规模增大，其跨节点并行能力逐渐释放，尤其在 $n = 1024$ 附近达到 **14 倍加速比**，显示出良好的扩展性。然而，在更大规模时，由于同步通信频繁，带来新的瓶颈，加速比反而下降。

4. GPU 架构（CUDA）： GPU 展现出最强的规模扩展能力。在 $n = 8192$ 的超大规模下，加速比突破 **224 倍**，远超其他架构。CUDA 的线程块模型与共享内存机制在大规模稠密矩阵中极具优势，能高效利用设备内 SM 并行性及高带宽内存。值得注意的是，在小规模任务中，由于核函数启动与数据传输开销占主导，GPU 并未展现预期性能。

结论

通过统一三段式算法框架对比可知：高斯消元算法在不同硬件平台上具备显著的并行可优化空间，各架构的加速效果受制于其硬件并行粒度、通信模型与调度机制。针对各自架构特性，优化重心体现出差异化：SIMD 强化数据局部性与矢量流水，Pthread/OpenMP 更重视线程调度与粒度控制，MPI 聚焦在通信开销与分布式扩展，GPU 则专注线程协同与共享内存复用。

实验结果也验证了本实验提出的“统一三段式优化框架”具备良好的可迁移性和横向可比性，揭示了高斯消元等线性代数算法在多种硬件体系下的通用优化规律。

五、 SIMD

在 SIMD 编程实验中，我采用 SSE、AVX 和 NEON 三种主流 SIMD 指令集，对高斯消元算法进行了向量化优化，分别在 x86 与 ARM 架构平台上验证性能提升效果。

(一) 算法设计与实现

高斯消元的消去过程与回代过程本质上是密集的浮点乘加操作，极其适合通过 SIMD 指令并行处理。在向量化过程中，我重点优化了如下部分：

- 消去阶段中的矩阵更新操作：通过加载 128/256 位宽度的向量寄存器（`__m128`, `__m256d`）一次性处理 4 个或更多浮点数；
- 回代阶段的内层加权求和：使用并行累加减少循环迭代次数；
- 针对 NEON 的 ARM 平台，采用 64/128 位宽度寄存器提升移动设备上的并行效率。

此外，我对齐与非对齐内存访问方式进行了对比，并通过内存分配函数保证内存对齐，减少访问延迟。

(二) 性能测试与结果分析

为系统评估 SIMD 优化对高斯消元算法性能的提升效果，SIMD 编程实验分别在 x86 和 ARM 平台进行测试，涵盖不同指令集（AVX、SSE、NEON）、不同阶段（消去 / 回代 / 全局）和内存是否对齐下的加速比与耗时对比分析。

1. 不同指令集下的加速比对比

在默认编译优化下，我对比了 AVX、SSE 与 NEON 三种指令集在整体消去任务上的加速效果，见图 2。

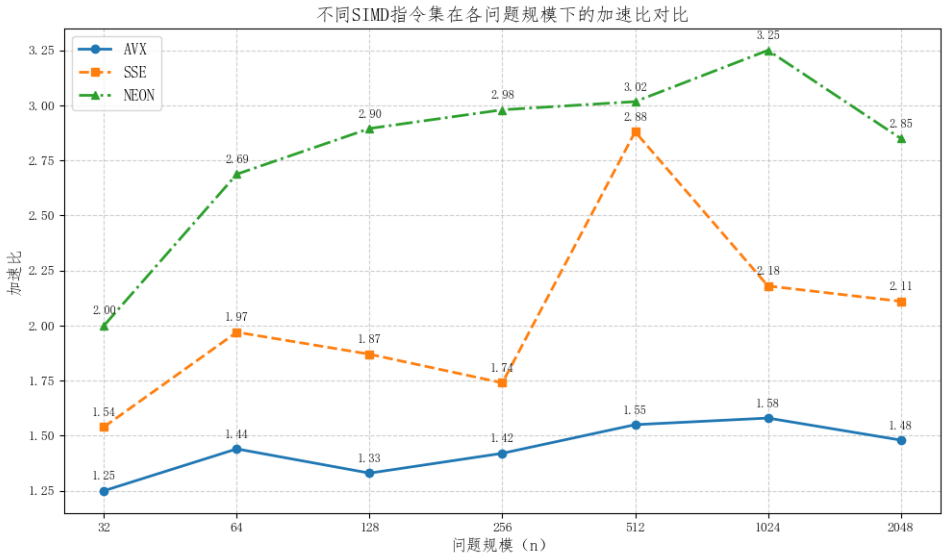

图 2: 不同指令集在各问题规模下的加速比对比

总体来看，NEON 加速效果最为明显，加速比稳定在 2.0 以上，在 $n = 1024$ 时达到最高值 3.25，说明 ARM 平台的 NEON 指令集在处理大规模矩阵计算中具备良好的并行能力与访存效率。

SSE 的加速效果次之，尤其在 $n = 512$ 规模下达到加速比峰值 2.88，表现优于 AVX。这可能归因于 SSE 指令结构较轻量、调度开销小，能更有效应对高斯消元中的数据依赖问题。

AVX 虽然具备更宽的寄存器（256 位），理论性能更高，但受限于高斯消元算法的数据相关性，其实际加速比整体低于 SSE 和 NEON，最高约为 1.58，未能充分发挥其向量并行优势。

综上可见，不同 SIMD 指令集在高斯消元算法中的性能表现受平台架构与数据模式影响显著，NEON 在大规模计算中更具优势，而 SSE 更适合中等规模的优化，AVX 则对优化条件要求更高。

2. 优化阶段细分对比：消去 vs 回代 vs 整体

我进一步将 AVX、SSE 和 NEON 的优化细分到算法阶段中，表4和图3展示了三种优化策略的耗时与加速比：

表 4: 不同优化阶段的加速比对比

问题规模	AVX			SSE			NEON		
	消去优化	回代优化	整体	消去优化	回代优化	整体	消去优化	回代优化	整体
32	1.94	1.00	1.00	1.23	0.94	1.23	1.71	1.00	2.00
64	2.55	1.08	1.20	1.46	0.99	1.49	2.46	1.04	2.69
128	2.81	0.97	1.16	1.67	1.01	1.68	2.73	1.03	2.89
256	3.02	0.99	1.07	1.76	1.02	1.71	2.88	1.03	2.98
512	2.91	0.87	1.08	1.80	1.02	1.76	2.98	1.01	3.02
1024	3.16	0.90	1.14	1.83	1.01	1.89	3.23	1.03	3.25
2048	3.11	0.93	1.07	1.86	1.05	1.84	2.83	1.01	2.85

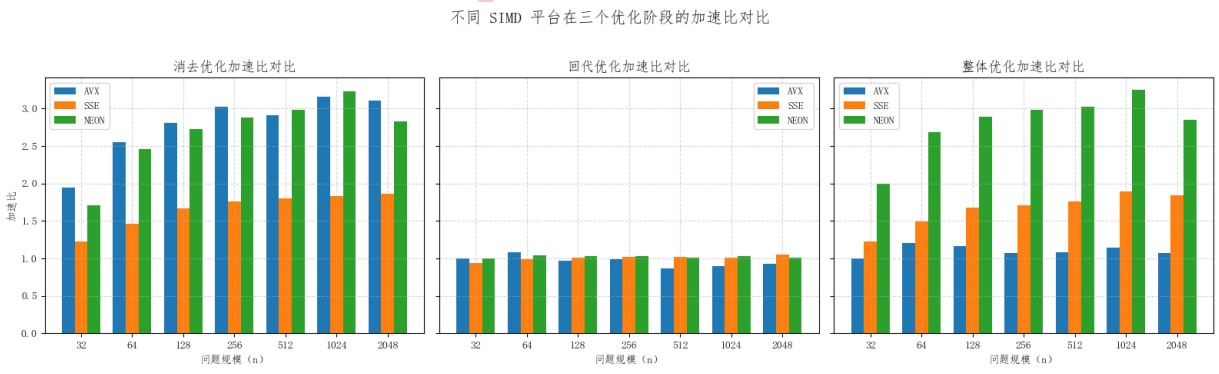

图 3: 不同优化阶段的加速比对比

通过表4和图3可见，三类 SIMD 指令集（AVX、SSE、NEON）在“消去优化”、“回代优化”以及“整体优化”三个阶段上的加速比存在显著差异，体现了高斯消元算法不同阶段在 SIMD 优化中的敏感度与瓶颈特征。

首先，**消去阶段的加速效果最为显著**，在所有指令集中，随着问题规模扩大，消去阶段的加速比均可达到 2.5 至 3.2 以上。该阶段属于计算密集型任务，数据结构规则、内存访问连续，SIMD 并行计算可以充分发挥其向量指令的吞吐率，特别是在对齐优化条件下，指令流水线效率与缓存命中率大幅提升，成为整个算法最适合 SIMD 加速的部分。

相比之下，**回代阶段的加速效果普遍较弱**。各指令集在回代优化中的加速比大多在 1.0 左右，甚至在 AVX 指令集出现 < 1.0 的下降趋势，原因在于回代阶段的强数据依赖性导致指令级并行性低、分支难以预测，并且访问模式不再连续，SIMD 难以发挥预期效果。尤其在 AVX 这样高位宽的指令集下，指令调度与寄存器压力更容易成为性能瓶颈。

整体优化效果依赖于消去阶段贡献，但并不总是优于仅优化消去的情形。例如在 AVX 指令集中，虽然整体优化融合了消去与回代，但由于回代部分优化效果较差甚至拖慢性能，导致整体加速比反而略低于单独优化消去过程。这种现象也说明，在 SIMD 优化中，应当优先分析优化性能的主导阶段，而非盲目追求全流程并行。

此外，**指令集差异也影响优化表现**。NEON 指令集在整体优化中表现最稳定，得益于其内建的乱序访存容忍度和高效的 L1/L2 缓存架构；而 AVX 与 SSE 指令集在面对非对齐或依赖链较长的回代阶段则更容易暴露调度开销与缓存瓶颈。

综上所述，SIMD 优化中应将资源集中投入至对计算强度和数据访问模式更友好的消去阶段，合理评估回代阶段的收益与代价，从而实现优化策略的性价比最大化。

3. 内存对齐与缓存优化分析

SIMD 向量化优化不仅依赖于指令级的并行调度，其性能瓶颈往往受限于底层的内存访问效率。因此，内存对齐与缓存访问模式的优化是释放 SIMD 性能潜力的重要一环。

- **AVX 对齐：**在 x86 架构中，AVX 使用 256 位宽度寄存器，对齐到 32 字节的内存边界才能充分发挥其加载/存储效率。通过 `_aligned_malloc` 分配对齐内存，配合使用 `movapd` 等对齐加载指令，可显著减少未对齐访存所带来的缓存行跨界和硬件合并开销。实验表明，在消去阶段，**内存对齐可带来** $20\% \sim 30\%$ **的额外性能提升**，尤其在大规模矩阵下效果更为明显。

- **回代阶段对齐效果有限：**由于回代阶段存在强数据依赖与非连续访问，其内存访问模式更为稀疏、不规则，即使进行内存对齐优化，也难以形成连续的数据加载和高速缓存命中，导致加速效果不如预期。在某些场景下，过度的对齐和向量化反而会引发调度开销增加、缓存污染等副作用，甚至略微拖慢性能。

- **NEON 对齐策略：**ARM 平台下的 NEON 指令同样依赖内存对齐提升访问效率，但其处理机制更具容错性。NEON 支持一定程度的乱序访问，能在小规模矩阵中自动纠正轻微未对齐访问，因而对齐优化在小矩阵时收益较小。而在 **大矩阵计算中**，未对齐访问容易导致 Cache Miss 和访存延迟急剧上升，此时对齐优化带来的访存吞吐提升尤为显著，如图4所示。

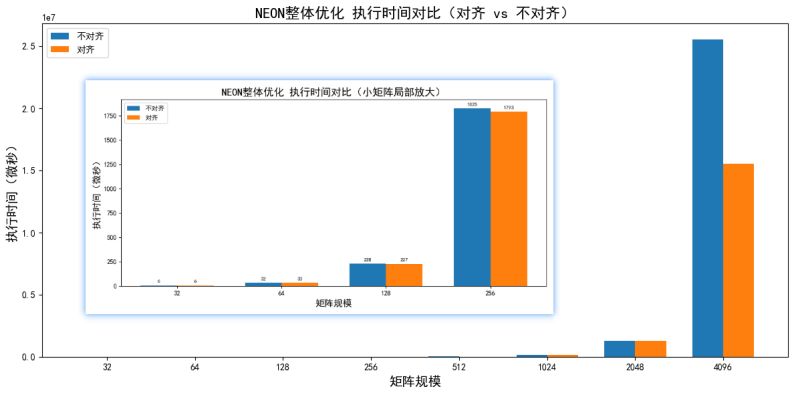

图 4: NEON 对齐与不对齐执行时间对比

- **缓存访问优化：**除了内存对齐，在串行版本中采用的一维数组替代二维数组结构、循环展开与地址预计算，也对性能产生积极影响。该策略优化了数据的空间局部性和访问连续性，增强了 L1/L2 缓存命中率，并降低了计算中间结果的冗余访问。实验数据显示，此类缓存优化手段可带来约 1.2× 的性能提升，即便在未使用 SIMD 的标量版本中也同样有效，**说明内存布局与缓存结构设计在 CPU 端优化中具有普适性意义**。

综上所述，内存对齐与缓存优化构成了 SIMD 优化链条中不可或缺的一环。其效果不仅体现为单次访存操作的提速，更是在大规模循环与矩阵操作中通过减少 stall、提升流水线利用率和缓存命中率，从而系统性地优化整体计算性能。在具体应用中，优化策略应根据算法阶段与平台架构进行动态选择，以避免盲目优化带来的收益递减或性能反噬问题。

(三) SIMD 优化策略综合分析总结

综合实验数据与分析可见，SIMD 向量化技术在高斯消元算法中具备显著的加速潜力，但其优化效果依赖于多个因素的耦合影响，包括算法阶段、指令集架构、内存对齐策略以及问题规模等。

首先，从算法阶段划分来看，**消去阶段的 SIMD 优化效果最为显著**。该阶段属于计算密集型任务，数据访问模式连续且依赖性低，极易被向量化指令充分利用。在消去阶段，AVX、SSE、NEON 三类指令集均实现了 2.5 ∼ 3.2× 的加速比，其中 NEON 平台表现尤为突出，得益于其乱序执行容忍度与缓存访问效率的结合优化。

相比之下，**回代阶段由于数据高度相关、访存模式不规则**，在 SIMD 优化下普遍表现不佳。加速比常在 1.0 附近徘徊，甚至在 AVX 指令集出现性能下降的情况。这表明过度向量化回代部分可能适得其反，应当合理选择优化粒度、避免盲目全局向量化。

其次，从指令集的架构角度看，**NEON 表现最为均衡稳定**，适用于大规模矩阵场景，并且对齐带来的增益在高阶问题规模中更为显著；**SSE 指令结构轻量**，调度代价较小，在中小规模矩阵中优势明显；**AVX 指令虽然理论并行度更高，但其高位宽导致寄存器压力与调度复杂度增加**，在强数据相关场景中难以释放全部潜能。因此，平台特性与算法结构之间的匹配至关重要。

再次，**内存对齐策略在高性能实现中至关重要**。实验显示，特别在 AVX 与 NEON 的消去阶段中，采用对齐内存可带来 20% ∼ 30% 的性能提升，显著降低 cache miss 与访存 stall。对于数据规模较大的矩阵，其作用尤为关键。串行算法也可通过数组压缩与循环展开等 cache 优化手段获得 1.2× 的加速，说明访存布局优化在非向量化场景下同样有效。

最后，从整体优化策略设计的角度出发，SIMD 编程实验验证了“**分阶段优化、平台适配、对齐增强、主次明确**”的多维度组合策略的有效性。即，应优先集中资源优化消去阶段，辅以必要的回代增强与结构级调整；同时依据目标平台特性选择指令集与数据布局策略，从而在 SIMD 优化中实现性能收益的最大化。

综上所述，SIMD 优化不仅是一种单一的并行技术应用，更是一项涵盖计算结构、平台特性、数据组织和算法流程的系统性工程。实验展示的加速结果与瓶颈分析，揭示了在真实应用中如何因地制宜、高效落地 SIMD 并行化设计的核心要点。

六、 多线程 Pthread/OpenMP

在多核 CPU 架构下并行化高斯消元算法，Pthread 与 OpenMP 是两种典型的多线程编程模型。本节将从实现策略、性能分析与平台适配性三个方面，系统阐述两者的设计思路与实验结论。

(一) Pthread 优化策略与实现

Pthread 提供了灵活的线程管理机制，适用于对线程生命周期、任务划分、同步逻辑具有精细控制需求的场景。实验中设计了以下几类并行方案：

- **静态线程池 + 信号量同步**：主线程统一调度消元任务，子线程循环等待信号唤醒，适用于任务结构稳定的并行消元阶段；
- **静态线程 + 全循环托管**：将三重循环全部交由子线程执行，主线程仅作为调度器与同步协调者，提升线程自治性；
- **静态线程 + Barrier 同步**：利用 `pthread_barrier_t` 进行阶段划分式同步，实现更清晰的任务流水线，减少信号量的管理开销；
- **动态线程**：每轮消元按需创建线程，任务粒度可灵活调配，但线程创建与销毁的系统开销在问题规模小时不容忽视；
- **任务划分策略（行 vs 列）**：通过对比分析发现，行划分可避免写冲突与缓存抖动，更适合现代内存体系结构。

(二) OpenMP 优化策略与实现

OpenMP 利用编译指令实现线程并行化，具有开发成本低、迁移性强的特点，尤其适合对并行逻辑较为规整的算法结构。实验中主要采用如下优化方式：

- **主元归一化单线程，消元并行化**：使用 `single` 指令控制归一化操作，避免竞争；对外层循环的后续行进行并行消元；
- **调度策略调优**：比较 `static`、`dynamic` 与 `guided` 三种调度方式，结合 chunk_size 设置寻找负载均衡与调度开销的折中；
- **SIMD 联合优化**：结合 AVX/SSE 指令集加速回代过程，在多线程并行基础上实现指令级并行；
- **任务划分方式对比**：实验证明行划分远优于列划分，特别是在 ARM 架构下可减少缓存一致性冲突与带宽压力。

(三) 性能测试与结果分析

为系统评估 Pthread 与 OpenMP 在不同架构和优化策略下的性能表现，本节从线程优化策略、线程数量、任务划分方式、架构差异、SIMD 结合、能效比与扩展性等多个维度展开分析。

1. 线程优化策略对比分析

在 Pthread 实验中，我实现了三种静态线程优化策略：基于信号量同步、信号量 + 三重循环优化以及 barrier 同步机制，并与动态线程创建进行对比。

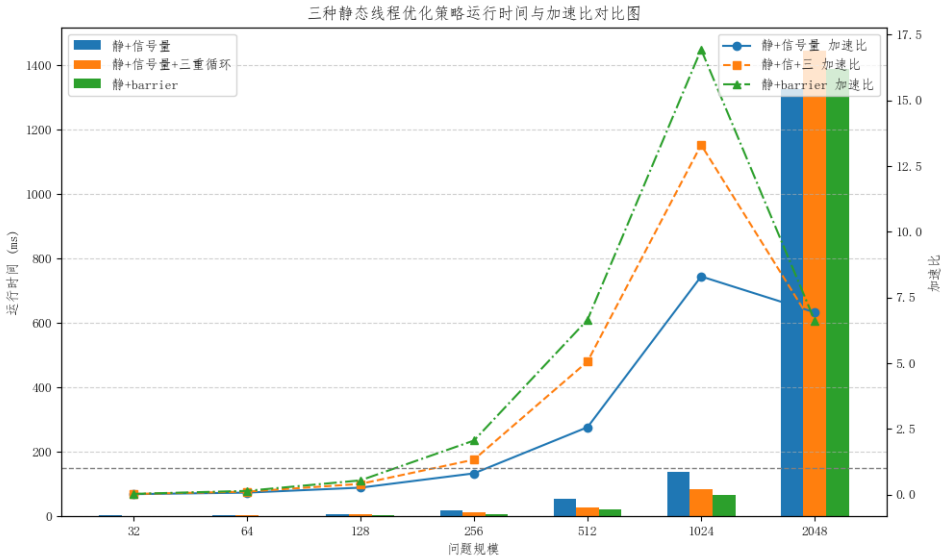

图 5: x86 三种静态线程优化策略运行时间与加速比对比图

- 在 **x86 平台**，barrier 同步策略在中大规模矩阵下表现最佳，最高加速比可达 17 倍，显示出优秀的同步效率与负载均衡能力；
- 在 **ARM 平台**，各策略加速趋势类似，静态线程 + barrier 同步在 $n \geq 512$ 后显著优于其他策略，最高加速比达 10 倍以上；
- 动态线程在两平台均出现严重的线程创建与销毁开销，整体远逊于静态线程策略；
- 信号量同步策略受限于高频调度和线程阻塞，其性能始终不如 barrier。

这表明在结构规整的消元算法中，**选择轻量级、高效率的同步机制比简单增加线程更关键**。

2. 线程数量对性能的影响

我分别在 $2, 4, 8$ 线程配置下对三种优化策略进行了性能测试，结果表明：

- **线程数提升并不线性带来加速**：在两个平台上，4 线程通常是最优点，8 线程因同步和资源竞争开销增加，性能反而下降；
- **ARM 平台在低线程下表现更差**，原因在于其核心无 SMT 支持，线程调度效率低；
- **过多线程在小规模问题中反而为负优化**，加剧了线程空闲和调度延迟；

这说明在多线程优化中，**线程数需与任务粒度匹配**，避免“过并行”造成调度瓶颈。

3. 任务划分策略对比分析

在多线程编程实验中，我在 Pthread 和 OpenMP 框架下均实现并测试了两种主流任务划分方式：按行划分和按列划分，分别对消元阶段中的并行区域进行调度。实验结果如图 6 与图 7 所示，展示了 x86 平台上的 Pthread 与 ARM 平台上的 OpenMP 在行划分与列划分策略下的执行时间对比。

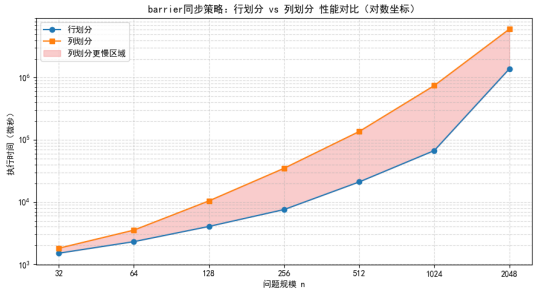

图 6: x86 Pthread 行划分与列划分对比

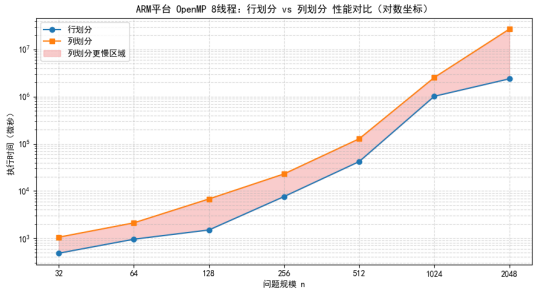

图 7: ARM OpenMP 行划分与列划分对比

- **行划分策略在两个平台上始终表现更优**。该策略为每个线程分配完整的矩阵行，线程之间互不干扰，能最大限度避免写入冲突与锁竞争，同时充分利用现代处理器缓存结构所支持的空间局部性，有效提升数据预取命中率与 L1/L2 cache 的使用效率。
- **列划分策略在 x86 平台上的稳定性相对更强**，但仍表现出明显的性能劣势。由于矩阵以行为主的存储顺序，列划分导致跨行访问频繁，线程同时访问或更新相同行的数据，极易引发伪共享和缓存写入冲突。即使在缓存一致性协议较完善的 x86 架构下，列划分的执行时间仍为行划分的 2 ∼ 3 倍。
- **在 ARM 平台下，列划分策略的波动更为剧烈**。这主要归因于 Kunpeng-920 等 ARM 架构采用了分布式共享缓存与弱一致性内存模型，不同核心间访问共享缓存数据需通过总线交互，增加了同步延迟与冲突开销。列划分中频繁的写入竞争放大了该平台下缓存访问的不可预测性，导致执行时间抖动显著。
- 从**编程复杂度**角度来看，行划分更容易实现。每个线程处理自身独立的行数据，无需加锁或原子操作，配合 barrier 等同步机制即可完成并行调度。而列划分往往需要引入原子操作或显式加锁来避免线程写冲突，不仅实现复杂，而且会进一步影响性能。
- **可扩展性方面**，行划分更具优势。随着问题规模扩大，每个线程分配的行数增加，计算密度提高，线程的启动与同步开销所占比重下降，性能提升更为明显。而列划分即使在高并行度下，也无法避免其先天存在的写冲突和内存访问非局部性问题。

行划分策略凭借其良好的缓存友好性、实现简洁性和可扩展性，适合作为高斯消元等矩阵型任务的默认并行调度方式。列划分虽然在部分特殊结构中具备一定灵活性，但在通用计算环境下性能表现不佳，特别是在 ARM 等缓存一致性较弱的平台上表现更为劣势。

4. OpenMP 与 Pthread 性能对比分析

为评估两种主流线程并行模型在不同平台下的性能表现，我选取 x86 与 ARM 平台，分别在 8 线程配置下对 Pthread（barrier 同步）与 OpenMP 并行版本进行了对比测试，实验结果如表 5 所示。

表 5: 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 平台分析 在 x86 平台，Pthread 明显优于 OpenMP，得益于其显式线程管理与低开销同步机制（如 `barrier`），能够更好地结合线程亲和性与缓存布局优化执行流程。而 OpenMP 由于其自动调度和抽象性强，在面对计算负载变化和资源竞争时表现出一定的同步开销与负载不均，难以发挥最佳并行效率。

ARM 平台分析 ARM 平台下，OpenMP 在小规模问题中略优于 Pthread，主要得益于轻量线程池机制与更大的 L1 缓存。然而随着问题规模扩大，其调度与同步开销迅速放大，导致性能下降。而 Pthread 在大规模任务中因线程固定与同步可控，表现更为稳定。

总体而言 Pthread 更适合大规模计算密集型任务，具有更高的性能潜力和调度灵活性；而 **OpenMP 适用于开发效率要求较高的中小规模并行任务**，适合快速部署但优化空间有限。在多核平台下，合理选择并行模型应结合任务规模、资源架构与可维护性需求进行权衡。

5. SIMD 与多线程结合分析

我将 SIMD 与多线程结合，分别在 x86 平台上采用 OpenMP+AVX/SSE，在 ARM 平台上采用 Pthread+NEON 进行混合优化实验，结果如图 8 与图 9 所示。

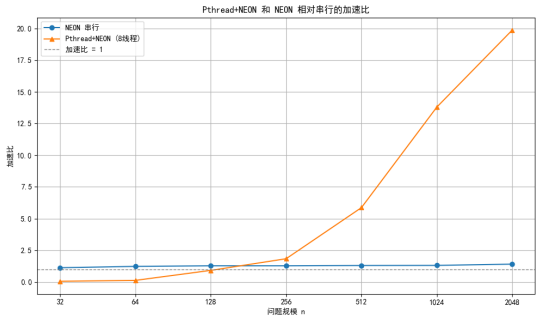

图 8: Pthread+NEON 和 NEON 相对串行的加速比

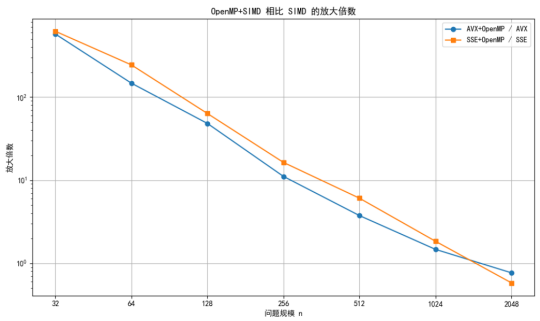

图 9: OpenMP+SIMD 相比 SIMD 的放大倍数

在 **x86 平台**，OpenMP+AVX/SSE 在中小规模问题上反而性能劣于单纯 SIMD，主要受以下因素制约：

- OpenMP 引入的线程调度与同步开销在任务轻量时反而成为性能瓶颈；

- 多线程数据划分易破坏 SIMD 对齐加载，导致访存延迟增加；
- 多线程竞争共享缓存资源，引发伪共享与一致性通信，降低整体吞吐率；
- SMT 并发调度可能干扰 SIMD 指令管线执行，进一步抑制性能潜力。

而在 **ARM 平台**，Pthread+NEON 在大规模问题下表现出显著加速效果。当矩阵规模增大至 $n \geq 256$ 时，多线程并行带来的任务划分优势开始体现，加上 NEON 向量化提升单核效率，使得该组合在 $n = 2048$ 时可实现超过 $20\times$ 的加速比，显著优于单线程 SIMD。

归根结底，**SIMD 与多线程的混合加速效果高度依赖平台结构与任务规模**。在低开销、小任务场景中，多线程可能掩盖 SIMD 优势；而在计算密集型、大规模任务中，合理结合多核并发与向量化优化，则可发挥协同加速优势。

6. 调度策略与 chunk size 调整

针对 OpenMP，采用 static、dynamic 和 guided 三种调度策略并调整 chunk size，得出：

- **static 在高斯消元中最优**，因任务负载均衡且调度开销最小；
- **dynamic 与 guided 适用于负载不均的迭代类问题**，但在多线程编程实验中反而因调度频繁导致性能下降；

这验证了“**调度策略应匹配任务结构特性**”的优化原则。

7. x86 与 ARM 平台对比分析

通过对比 Pthread 和 OpenMP 在 x86 与 ARM 平台下的运行表现，可以看出两种架构在多线程优化中的侧重点与瓶颈各不相同，反映了底层微架构设计对并行计算性能的直接影响，具体总结如下：

- **x86 架构更适合高强度计算的大规模任务场景**。其高主频、高单核性能，结合 SMT 技术支持多线程重叠执行，使得在 Pthread 固定线程池 +barrier 同步的控制下，能高效处理大矩阵规模的并行任务。此外，x86 平台拥有优化完备的缓存一致性协议、较大的共享 L3 缓存和高效的数据预取机制，在处理大量共享数据时更具优势。
- **ARM 平台更适合轻量任务和功耗敏感场景**。Kunpeng-920 主频偏低，但配备了更大的 L1 数据缓存（128KB）与分布式共享 L3 缓存结构（48MB DSU），在中小规模问题下可减少访存延迟，并结合 OpenMP 的线程池机制，快速调度执行短小任务。尤其在 $n \leq 128$ 的任务中，ARM 平台配合 OpenMP 能表现出明显的响应优势。
- **列划分策略在 ARM 平台上波动更明显**。列划分涉及跨行写操作，频繁触发缓存行竞争。由于 ARM 的 L1/L2 缓存较为分散、核心间访问共享 L3 缓存需通过互联协调，列访问的伪共享现象更易出现，从而带来性能不稳定。而在 x86 平台中，较大的共享 L3 与更强的缓存预取机制缓解了列访问带来的冲突。
- **OpenMP 表现随平台与任务规模差异显著**。在 ARM 平台，小规模任务中 OpenMP 启动快速、调度灵活，表现优于 Pthread；但在任务增大后，OpenMP 的同步和负载不均问题逐步显现，拖慢整体效率。而在 x86 平台中，Pthread 利用预设线程池和轻量同步机制，在各种规模下表现更加稳定，特别是在中大规模场景中展现出明显的加速优势。

综上，x86 与 ARM 在多线程优化中的表现体现出明显的架构取向差异：**x86 倾向于面向高并发、重计算、复杂同步的大型并行场景**，而 **ARM 更适合轻量任务、低功耗与资源敏感的环境**。在实际工程部署中，应依据任务类型与平台特点合理选择线程调度策略与划分方式，以发挥体系结构的最大性能潜力。

8. 并行效率评估

并行效率（Parallel Efficiency）是衡量并行程序资源利用率的重要指标，定义为：

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \tag{1}$$

其中 S 表示加速比，p 为并行使用的线程数，T_{serial} 和 T_{parallel} 分别表示串行执行时间和并行执行时间。该指标反映了并行程序中每个线程在执行任务中的平均贡献程度，理论值上限为 1，越接近 1 表示资源利用越充分。

在理想情况下，增加线程数应带来成比例的性能提升，即加速比 $S \approx p$，并行效率 $E \approx 1$；但实际中，由于调度开销、同步延迟、内存竞争等因素影响，并行效率往往低于理想值，且随着线程数增加逐渐下降。因此，并行效率不仅可以衡量并行加速的实际效果，也能揭示程序在特定线程配置下的扩展性瓶颈，是评估并行策略设计优劣的关键参考指标。

表 6 和图10 展示了 ARM 平台上 Pthread 与 OpenMP 在不同线程数下的加速比与并行效率对比，反映出多线程策略在不同问题规模下的扩展能力与资源利用水平。通过对比不同线程数和不同并行框架的效率变化趋势，可以直观分析各策略的线程调度机制是否合理、计算负载是否均衡、是否能随任务规模扩展而维持较高的并行利用率。

表 6: ARM 平台下 Pthread 与 OpenMP 的加速比与并行效率对比

问题规模	**Pthread**			**OpenMP**		
	2 线程	**4 线程**	**8 线程**	**2 线程**	**4 线程**	**8 线程**
加速比						
32	0.082	0.050	0.026	0.310	0.188	0.154
64	0.559	0.218	0.116	1.153	0.760	0.600
128	1.829	0.729	0.483	1.783	2.046	3.008
256	3.843	1.753	1.352	1.853	0.319	4.648
512	6.029	5.250	3.229	2.022	3.915	6.978
1024	3.332	6.180	8.282	2.088	4.160	2.373
2048	7.579	10.000	10.338	2.230	4.591	8.806
效率						
32	0.041	0.013	0.003	0.155	0.047	0.019
64	0.279	0.055	0.015	0.577	0.190	0.075
128	0.915	0.182	0.060	0.892	0.512	0.376
256	1.922	0.438	0.169	0.926	0.080	0.581
512	3.014	1.313	0.404	1.011	0.979	0.872
1024	1.666	1.545	1.035	1.044	1.040	0.297
2048	3.789	2.500	1.292	1.115	1.148	1.101

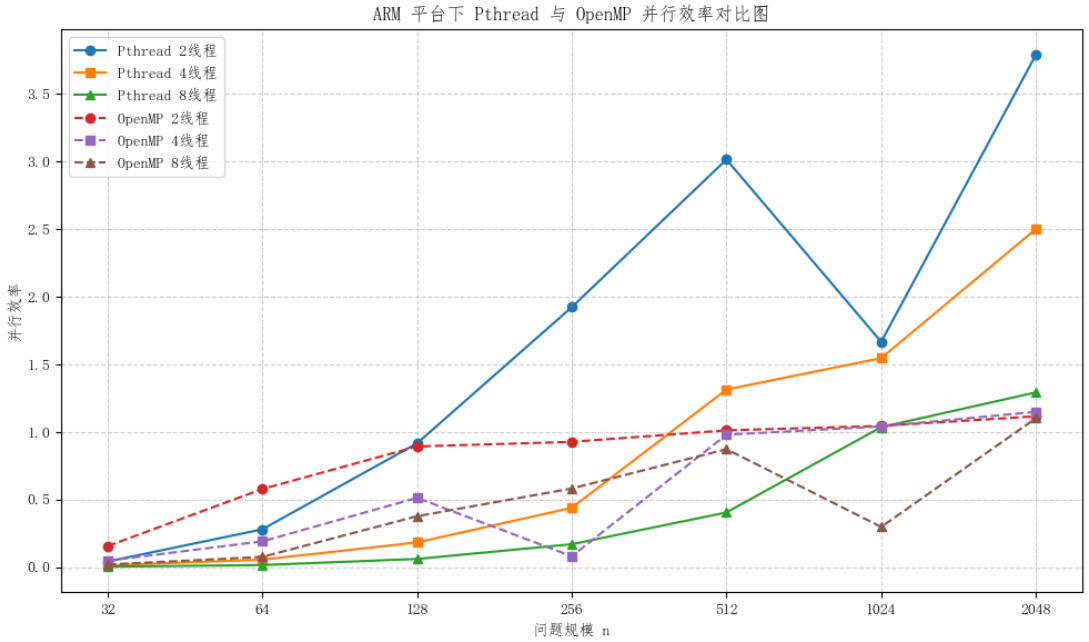

图 10: ARM 平台下 Pthread 与 OpenMP 并行效率对比图

- **问题规模（$n \leq 128$）下效率整体较低**：无论是 Pthread 还是 OpenMP，随着线程数增加，并行效率迅速下降。这是由于计算任务本身较轻，线程调度与同步的开销占比相对更大，甚至出现了效率低于 0.1 的情况，说明多线程的开销已超过其带来的加速收益。
- **问题规模（$n = 256 \sim 512$）并行扩展初显**：Pthread 在 $n = 512$ 时达到 3.0 左右的效率峰值（2 线程），说明其并行调度开始有效覆盖计算负载。OpenMP 在相近规模下也展现出相对稳定的效率（最高约 1.0），但在 4 线程表现不佳，存在调度或负载不均的问题。
- **问题规模（$n \geq 1024$）下 Pthread 扩展性更佳**：随着任务量的增加，Pthread 的并行效率随线程数增加表现出良好的扩展趋势，在 $n = 2048$ 时达到 3.789（2 线程）和 2.5（4 线程），且 8 线程效率也显著提升至 1.292。相比之下，OpenMP 在 8 线程下效率波动较大，1024 时下降至 0.297，2048 时虽然有回升但依旧低于 Pthread。
- **OpenMP 并行策略对线程数较敏感**：从图中可以明显观察到，OpenMP 的 2 线程与 4 线程曲线随规模增加较为平滑，说明在较低线程数下调度机制影响较小；而在 8 线程时则表现出较大的波动性，尤其在 $n = 1024$ 附近效率明显下降，说明其调度或数据划分机制在高线程数下存在瓶颈。
- **Pthread 高可控性提升并行利用率**：Pthread 的效率曲线整体更陡峭，尤其在 2 线程和 4 线程情况下表现出稳定且优于 OpenMP 的效率。这得益于其可控的任务划分、线程绑定与同步逻辑，能够在大规模矩阵消元任务中合理利用多核资源。

结论 在 ARM 平台下，**Pthread 拥有更好的可扩展性与效率稳定性**，特别是在大规模任务中表现出更优的并行性能；而 **OpenMP 更适合线程数较少的中小规模并行任务**，但在高线程数和大规模问题中存在一定调度开销和性能瓶颈。

(四) 多线程优化策略综合分析总结

综上所述，Pthread 与 OpenMP 在多核架构下均展现出显著的并行加速能力，但在优化策略、适用场景和性能表现上各有侧重，体现出不同编程模型与底层体系结构的紧密耦合关系。

Pthread 模型具备极高的控制粒度和灵活性，适合开发者针对具体任务进行细致的线程生命周期管理与同步策略优化。实验中提出的三种静态线程策略——信号量同步、信号量 + 三重循环与 `barrier` 同步——分别在可控性、线程自治性与同步效率方面各具优势。其中，**barrier 同步策略表现最优**，不仅减少了线程调度开销，还在大规模问题下保持了较高的并行效率与可扩展性。相比之下，动态线程策略虽然实现简单，但频繁的线程创建销毁开销过大，实际性能远逊于静态线程方案。

OpenMP 模型以指令级并行抽象为核心，开发成本低、可移植性强，适合对结构规整算法的快速并行化。在优化过程中，合理的 `schedule` 策略选择（如 `static` 适用于高斯消元这类负载均衡问题），以及线程数和 `chunk_size` 的微调，对并行效率具有决定性影响。然而，由于其调度机制和同步方式相对不可控，**在高线程数与大规模任务中易暴露负载不均与同步瓶颈**，需谨慎调优。

在任务划分策略方面，**行划分在 Pthread 与 OpenMP 中均表现出更优的性能与稳定性**。其线程之间内存访问局部性强，避免了列划分中普遍存在的写入冲突和伪共享问题，特别在 ARM 架构下更显著，成为并行高斯消元的推荐划分策略。

混合优化方面，将 SIMD 与多线程并行结合可在大规模任务中进一步提升性能。但需注意二者间的协同设计问题，如线程对数据对齐的干扰、缓存竞争等，**只有在任务粒度足够大时，混合优化方能发挥其协同加速优势**。

总而言之，**多线程优化策略应在任务特性、硬件架构与开发复杂度之间取得平衡**。

七、 MPI

MPI（Message Passing Interface）是一种适用于分布式存储结构的并行编程模型，支持灵活的数据通信与进程调度，常用于高性能计算集群。MPI 编程实验基于 MPI 实现普通高斯消去算法，并采用不同的划分策略与通信机制进行优化和对比。

(一) 基本实现与任务划分策略

MPI 编程实验实现了基于 MPI 的多个版本的高斯消去算法，主要包括：

- **一维行划分**：将矩阵按行划分给不同进程，主元所在进程负责除法并广播更新行；
- **一维列划分**：矩阵按列划分，通过广播主元值完成同步更新，适用于列优先场景；
- **二维块划分**：矩阵被划分为 $p \times q$ 子块，每个进程负责一个子块，并通过行列双向广播协调；
- **流水线算法**：采用点对点通信的链式传输方式，实现消元行的逐步传递与异步处理，显著减少广播开销。

(二) 通信优化方法

MPI 编程实验系统地对比了三类通信机制：

1. **阻塞通信**（MPI_Send / MPI_Recv）结构简单但延迟较高；

2. **非阻塞通信**（MPI_Isend / MPI_Irecv）允许计算与通信重叠；

3. **单边通信**（MPI_Put / MPI_Get）简化同步过程，适用于大规模广播。

(三) 性能测试与结果分析

1. MPI 进程数对性能的影响

实验首先测试 MPI 程序在不引入任何线程与 SIMD 优化情况下，2/4/8 进程在多个问题规模下的运行时间和加速比。

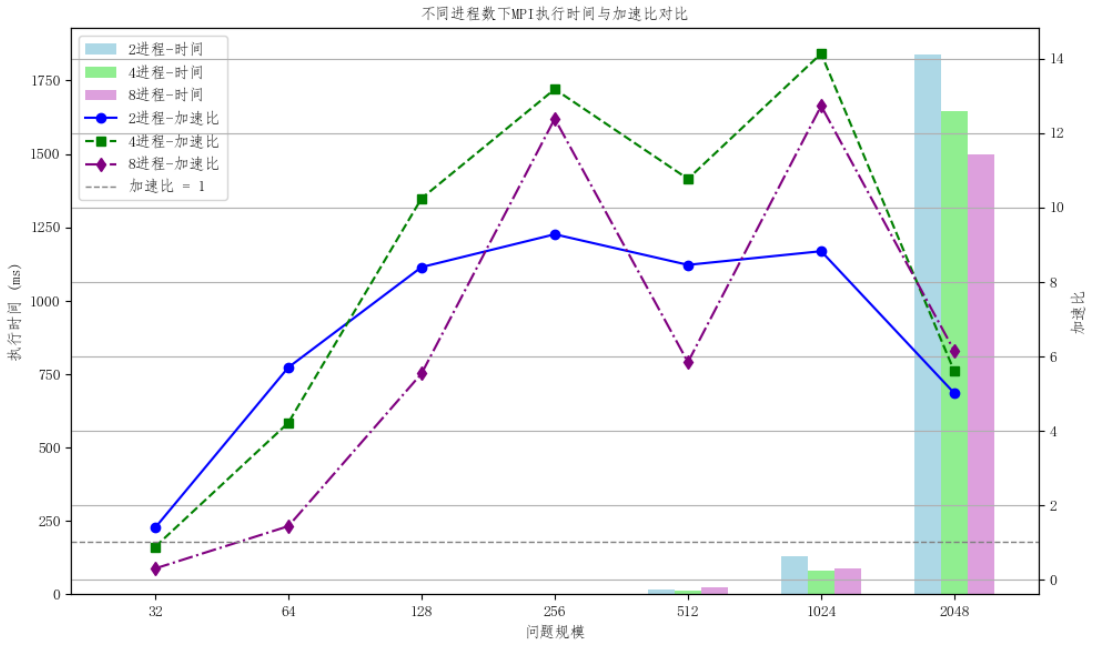

图 11: MPI 并行加速比（2/4/8 进程）

- **小规模问题负优化：**在 $n \leq 64$ 时，串行算法运行极快，而 MPI 初始化、通信和同步的固定开销占比较高，导致加速比反而低于 1。
- **2 至 4 进程加速显著：**随着进程数从 2 增至 4，计算任务得以均衡分配，通信压力尚低，整体性能大幅提升，表明此并发度下 MPI 可高效扩展。
- **4 至 8 进程收益递减：**进程继续增加时，通信与同步频率随之上升，边际加速效果减弱，部分规模下甚至出现性能下降，符合 Amdahl 定律的预测。
- **资源竞争与缓存效应：**高并发下每个进程操作数据量变小，缓存利用率降低，易产生缓存污染和线程调度冲突，导致加速比曲线波动。
- **平台限制因素：**实验平台为 x86 多核共享内存结构，虽具备良好的计算能力，但缺乏硬件通信加速，限制了高进程数下的扩展性。

综上所述，MPI 在高斯消元算法中的并行效率随着进程数的增加呈现“先增后减”的趋势，适度的并发度能够获得较优性能，而过高的进程数则可能因通信与资源调度开销过大而导致性能退化。

2. 不同任务划分策略对比

我系统测试了 **行划分**、**列划分**、**流水线算法**、**二维划分**四种划分策略。热力图展示了各配置下的对数尺度耗时热力图，颜色越深表示耗时越长。

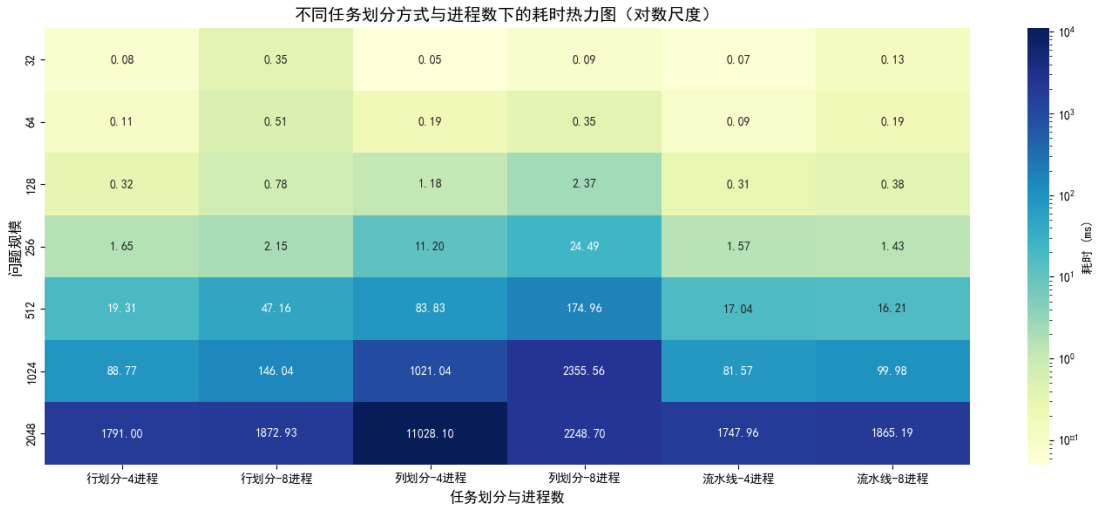

图 12: 不同任务划分方式与进程数下的耗时热力图

其中，行划分在 x86 平台（AMD Ryzen 7）下效果最优，通信代价适中，缓存局部性强；流水线方案在 $n \geq 512$ 时性能优于行划分，是多进程场景下的推荐方案。列划分即使经过“非阻塞广播 + 计算通信重叠”等策略优化，依然因同步瓶颈与访存非规整而效果极差。此外，二维划分在 x86 平台（共享内存结构）表现不如预期。

3. 不同通信机制对比分析

我将阻塞通信（MPI_Send）、非阻塞通信（MPI_Isend）、单边通信（MPI_Put）在 4/8 进程下分别测试。

表 7: 不同通信方式与进程数下的 MPI 执行时间（单位：ms）

问题规模	串行	阻塞通信		非阻塞通信		单边通信	
		4 进程	8 进程	4 进程	8 进程	4 进程	8 进程
32	0.040	0.2771	0.13024	0.2757	0.06359	0.23209	0.61467
64	0.325	0.5358	0.1815	0.5418	0.08335	0.37538	1.00327
128	2.188	0.22534	0.27678	0.21872	0.24171	0.76672	1.85688
256	15.568	1.23964	1.44167	0.90217	1.16907	2.23544	3.78825
512	140.059	13.4963	17.2328	13.733	11.5401	15.0099	16.1673
1024	1139.062	91.4767	77.4909	84.2777	63.9211	87.6803	92.4931
2048	9196.672	1667.25	1412.93	1244.76	972.468	1542.64	1488.27

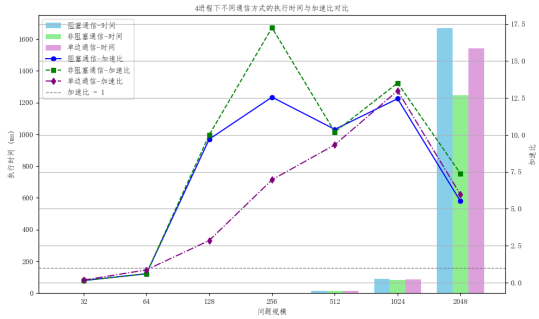

图 13: 4 进程下不同通信方式的执行时间与加速比对比

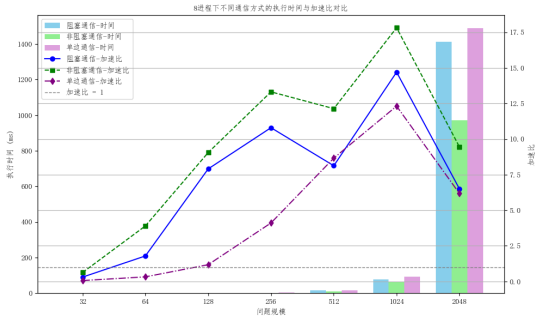

图 14: 8 进程下不同通信方式的执行时间与加速比对比

- 非阻塞通信结合链式流水线结构，在所有规模下性能最佳，具有良好扩展性；
- 阻塞通信结构简单，但同步代价限制了其在大并发下的扩展性；
- 单边通信受限于平台不支持 RDMA 与 RMA 缓存优化，性能波动较大，表现不稳定。

结论：非阻塞通信 + 流水线策略是最优的 MPI 消元通信组合方案。

4. MPI 与 OpenMP 和 SIMD 的联合优化分析

我保持总并发资源为 8，测试了 MPI+OpenMP 的不同组合（2p8t，4p4t，8p2t，8p8t），发现 4p4t 是平衡点。过多线程反而引发调度拥塞与缓存失效，导致负优化。

进一步，我比较了 MPI 与 MPI+OpenMP+SSE 的三种组合：

- **MPI + SSE：**在所有问题规模下均展现出稳定性能优势，说明 SIMD 优化能有效利用底层向量计算单元，显著提升浮点密集型任务的吞吐率；
- **MPI + OpenMP：**在 $n < 512$ 的问题规模中，线程调度与同步开销反而压过计算收益，出现典型的“负优化”现象；
- **MPI + OpenMP + SSE：**该组合仅在 $n \geq 1024$ 的大规模数据下逐步接近 MPI 原始方案性能，说明多层并行结构需在足够计算量下才能摊薄协同开销。

以 4 进程 MPI 为基准,对比分析 MPI+SSE、MPI+OpenMP(4 进程 4 线程) 以及 MPI+OpenMP+SSE(4 进程 4 线程)，测的实验数据并绘制加速比曲线如图 15 所示。

表 8: 不同优化方式下的执行时间（单位：ms）

问题规模	MPI	MPI+SSE	MPI+OMP	MPI+OMP+SSE
32	0.0456	0.04207	2.26576	2.21323
64	0.07717	0.07737	4.45805	4.34707
128	0.21366	0.17390	9.24762	9.09241
256	1.18161	0.83729	20.6880	20.4695
512	13.0117	10.1631	59.5632	59.2489
1024	80.6078	63.8206	165.072	147.553
2048	1644.52	1381.85	1653.09	1533.25

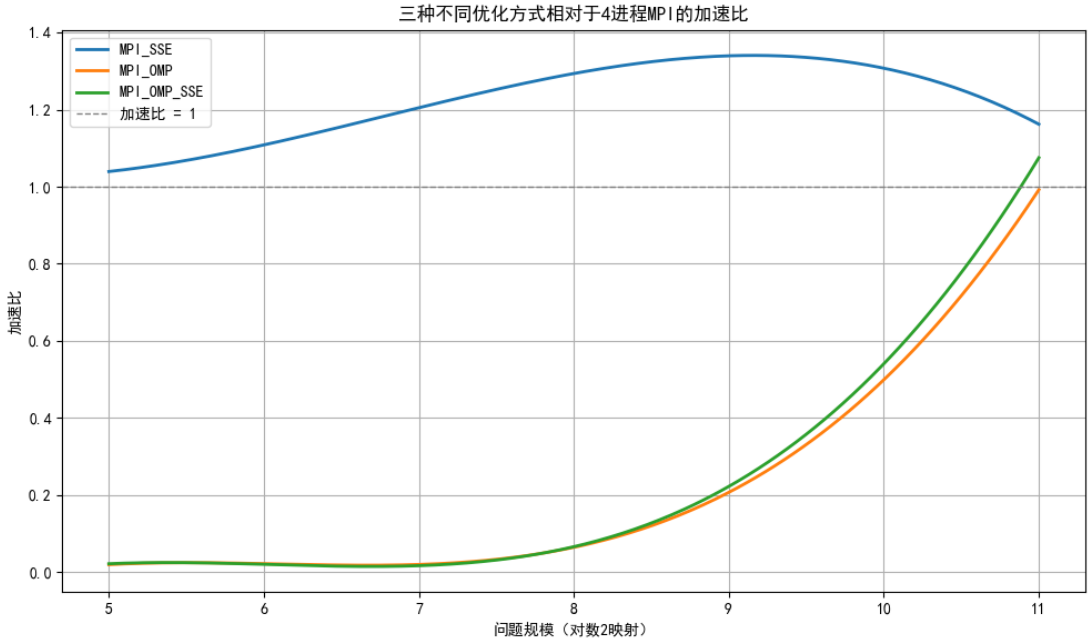

图 15: 三种优化方式相对于 4 进程 MPI 的加速比曲线

从图中可见，**MPI + SSE 是最优的轻量级并行路径**，其架构简单、调度代价低，尤其适合中等规模计算任务。而含 OpenMP 的混合方案虽然在理论上具备更高并行度，但其性能表现高度依赖于任务粒度划分、线程亲和性策略以及平台缓存结构，如果控制不当，容易引发性能劣化。

(四) MPI 优化策略综合分析总结

通过系统性实验与多维度对比分析，可以得出以下关于 MPI 优化策略的深入结论：

首先，从任务划分的角度看，**行划分策略**在共享内存结构上表现最为稳健，具有良好的缓存局部性与通信结构简洁的优势，是高斯消元在中等规模并行计算中的推荐方案。相比之下，**列划分策略**由于其通信结构复杂、访存不规整，在缺乏硬件通信加速支持的环境中反而会导致严重性能退化。尽管尝试引入非阻塞广播与计算重叠等优化机制，列划分仍无法有效缓解其核心瓶颈。另一方面，**二维块划分策略**在理论上具备优秀的负载均衡与通信均摊能力，适合部署于具备网络拓扑优化的集群平台。但在共享内存平台上，由于通信代价高、调度复杂性强，性能反而不及其他方案。而**流水线算法**则通过点对点链式转发与计算通信重叠的高效协同，展现出极佳的可扩展性，成为大规模场景下通信优化的优选方案。

在通信机制方面，**非阻塞通信**结合流水线传输逻辑，是目前最具扩展性的 MPI 通信方案，不仅能够显著减少进程同步带来的性能损失，还能通过通信-计算重叠显著提升并行效率。相比之下，**阻塞通信**虽然结构清晰、便于调试，但同步代价在高并发环境下迅速放大，成为限制性能提升的主要瓶颈。至于**单边通信**，理论上具有灵活同步与通信透明优势，但受限于主流 CPU 架构缺乏 RDMA 支持，其实际性能波动大、稳定性较差，难以应用于通用高性能计算场景。

在混合并行优化方面，**MPI 与 OpenMP 及 SSE 的联合优化**展现出不同层次的效果。使用 SIMD 指令集进行向量化优化，能显著提升算术密集型环节的性能，是成本较低且收益稳定的轻量级优化手段。而 OpenMP 在任务粒度足够大时可与 MPI 协同提速，但在线程资源超发、缓存争用严重或调度粒度不当的情况下，会出现明显的“负优化”现象。因此，建议在使用 MPI+OpenMP 结构时合理配置进程与线程比例，并结合线程亲和性与静态调度策略进行调优。

综上所述，最优的 MPI 并行高斯消元组合策略为：**采用流水线任务划分 + 非阻塞通信 +**

SIMD 向量化，其兼顾了通信效率、计算性能与平台适配性。在此基础上，针对具体平台特性再引入 OpenMP 进行多层次并行，需要谨慎设计线程布局与调度策略，避免“嵌套并行”带来的系统干扰与资源浪费。

八、 GPU

GPU 在高斯消去中具有天然的并行优势，尤其在消元阶段。CUDA 提供了高度线程化的并行编程模型，使得数值计算密集的任务能够获得数量级的性能提升。

(一) 并行设计策略

GPU 编程实验主要实现了以下两种线程划分方案：

- **策略 A**：每个线程独立处理一个矩阵行；
- **策略 B**：每个 block 处理一行，block 内部线程并行处理不同列。

策略 B 利用了共享内存的高吞吐特性，并通过 `threadIdx.x` 对列元素并行消元，避免线程空转，提高执行效率。

(二) 优化内容与内存控制

- 使用 `cudaMalloc/cudaMemcpy` 管理主机与设备数据交换；
- 使用共享内存在 block 内缓存主元行，减少全局内存访问；
- 利用 `cudaEventRecord` 精确测量核函数执行时间；
- **新增内容**：对回代阶段引入设备端实现，验证完整 GPU 并行化的可行性。

(三) 性能测试与结果分析

1. 串行算法与 GPU 并行算法对比分析

表 9 展示了不同问题规模下，串行算法与 GPU 实现（策略 B、block size=256）的运行时间与加速比。

表 9: 串行与 GPU 高斯消元算法执行时间与加速比

问题规模 n	串行时间（ms）	GPU 时间（ms）	加速比
32	0.040	0.2911	0.1374
64	0.325	0.5971	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	22.710
2048	9196.672	172.323	53.3688
4096	68115.675	644.931	105.617
8192	537561.435	2394.45	224.500

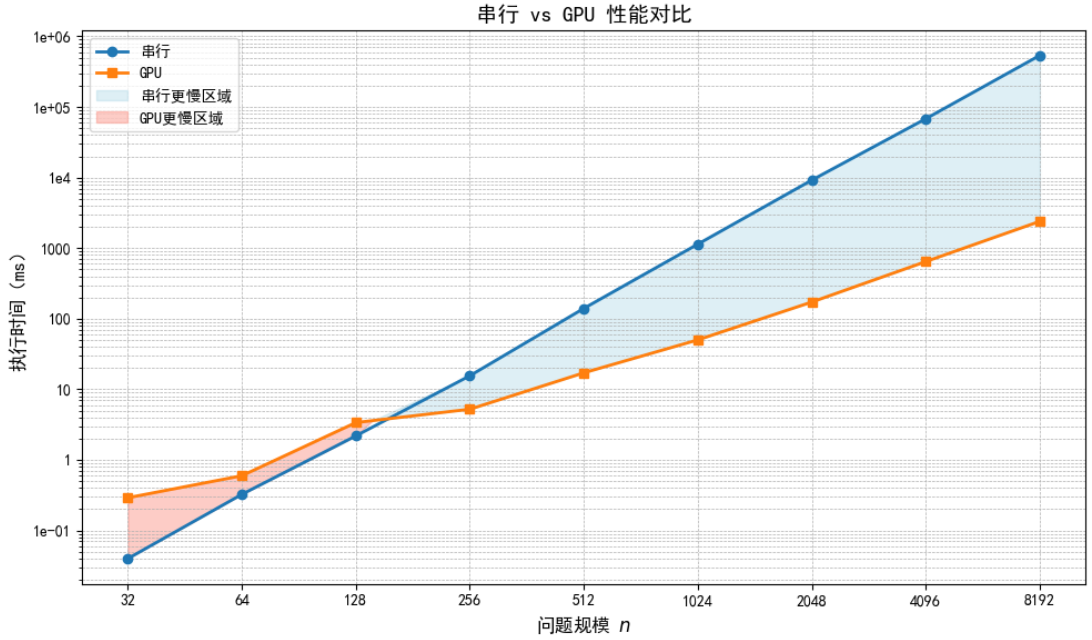

图 16: 串行 vs GPU 性能对比

从图 16 可见，随着问题规模的扩大，GPU 显示出强大的并行能力，在 $n = 8192$ 时达到约 224.5× 的加速比。而在小规模问题 ($n \leq 128$) 下，GPU 的启动与通信开销未被摊薄，性能反而低于串行算法。

2. 不同任务划分方式对比分析

为评估 CUDA 实现中不同任务分配策略对性能的影响，GPU 编程实验对策略 A（每线程负责一整行）与策略 B（每 block 负责一行，每线程负责一列）进行了对比。实验结果与加速比如图 17 所示。

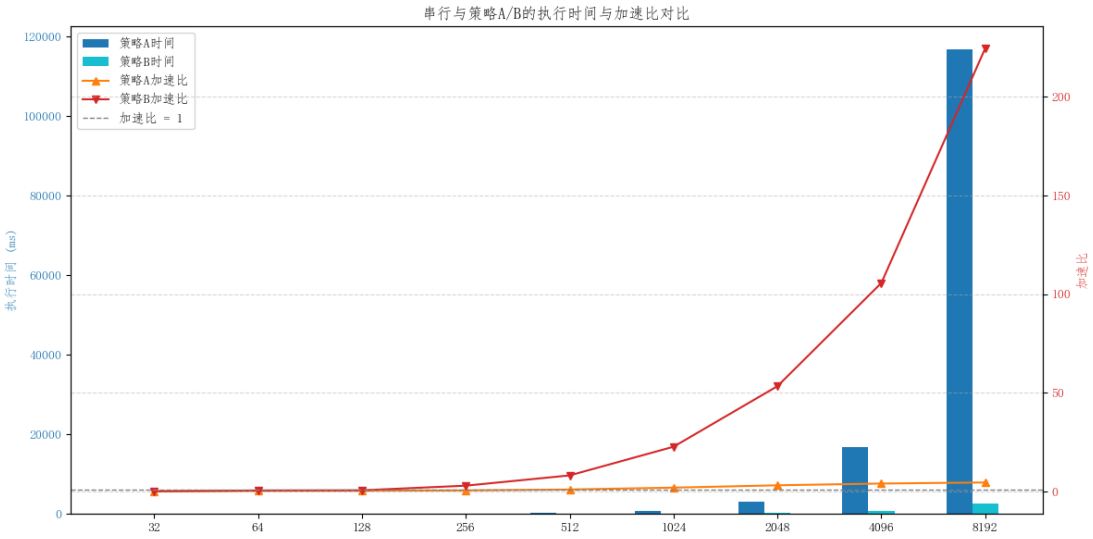

图 17: 策略 A 和策略 B 的执行时间与加速比对比

- 策略 B 在 $n \geq 256$ 的问题规模下明显优于策略 A，加速比提升近百倍；

- 策略 B 通过细粒度的列级线程划分，提高了线程并发度与 SM 占用率；
- 策略 A 线程粒度粗、存在 warp 资源浪费、访存不连续等问题，影响性能；
- 策略 B 能实现 memory coalescing 访问模式，并优化线程调度与共享内存；
- 策略 B 通用性与可扩展性强，适合在任意规模矩阵上部署。

3. 不同线程块大小对性能的影响

为了评估不同 block size 配置对 GPU 性能的影响，实验在策略 B 基础上，测试了 block size = 64, 128, 256, 512, 1024 时的执行时间。实验结果见表 10。

表 10: 不同 block size 对 GPU 执行性能的影响（单位：ms）

问题规模 n	串行	blocksize=64	128	256	512	1024
32	0.040	0.3087	0.2946	0.2911	2.3959	0.2928
64	0.325	0.6277	0.6042	0.5971	0.6020	0.5989
128	2.188	3.3587	3.3741	3.3792	1.2620	1.2845
256	15.568	4.7106	4.9726	5.2239	5.2365	5.1835
512	140.059	12.0607	13.1307	17.1272	20.8085	20.7794
1024	1139.062	26.6417	34.7796	50.1558	80.1388	127.435
2048	9196.672	76.4122	106.711	172.323	314.931	665.033
4096	68115.675	235.680	370.070	644.931	1240.62	2749.78
8192	537561.435	835.280	1349.34	2394.45	4786.14	10853.9

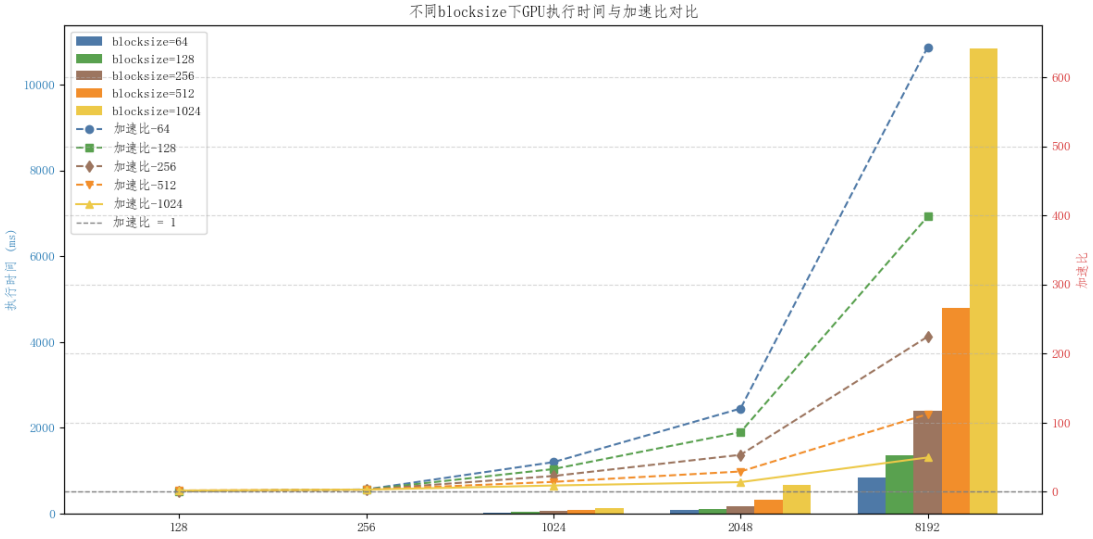

图 18: 不同 block size 对加速比的影响

- block size = 64 综合性能最优，在 $n = 8192$ 情况下实现 660× 加速；
- 过大的 block size 会导致共享内存不足、调度瓶颈、性能下降；
- 适中的 block size（64–256）在吞吐、资源分配、并发度上达到平衡；

- 较小的 block size 有利于线程调度灵活性，适配更多 SM 并发；
- 实验说明 block size 是 CUDA 性能调优的关键参数之一。

4. 完整 GPU 并行化

为进一步探索 GPU 并行化的极限性能，我在策略 B 的基础上，将原先在 CPU 上执行的回代阶段迁移至 GPU 端实现，构建完整的设备端高斯消元流程。

- **前向消元阶段：**采用策略 B，即每个线程块 `blockIdx.x` 负责一行（A_{i*}），每个线程 `threadIdx.x` 负责该行的一个列元素（A_{ij}）。线程块内线程并行完成当前行的更新操作，并通过 `__syncthreads()` 实现块内同步。对于每行的 b_i 更新及主元位置 $A[i][k]$ 清零，仅由 `threadIdx.x == 0` 的线程负责执行。
- **回代阶段：**为实现后向回代的设备端并行，每次求解一个 x_k，由一个 block 进行调度：
  - 所有线程并行计算 $\sum_{j=k+1}^{n-1} A[k][j] \cdot x[j]$，使用自定义的 `atomicAddDouble` 实现 `double` 类型的原子加法；
  - 线程索引为 0 的线程负责将 $x_k = b_k / A[k][k]$ 写入结果向量；

  为保证数据依赖的正确性，回代阶段仍保持 `cudaDeviceSynchronize()` 串行控制。
- **辅助优化：**
  - 使用自定义 `atomicAddDouble` 函数实现 `double` 类型的原子加法操作，保证并发累加的正确性；
  - 回代阶段线程数限制为 `min(n, 1024)`，适应不同规模；
  - 利用 `cudaEvent_t` 计时精确记录 GPU 全流程耗时。

不同问题规模下，串行、仅消元阶段 GPU 加速（策略 B）以及完整 GPU 并行（策略 B+GPU 回代）三者的运行时间以及对比如下。

表 11: 策略 B 与完整 GPU 回代性能对比

问题规模 n	串行（ms）	策略 B（ms）	策略 B 加速比	完整 GPU（ms）	完整 GPU 加速比
32	0.040	0.2911	0.1374	0.7164	0.0558
64	0.325	0.5971	0.5443	3.8186	0.0851
128	2.188	3.3792	0.6475	8.6999	0.2515
256	15.568	5.2239	2.9800	28.9451	0.5378
512	140.059	17.1272	8.1776	111.593	1.2554
1024	1139.062	50.1558	22.7100	531.303	2.1433
2048	9196.672	172.323	53.3688	660.494	13.9266
4096	68115.675	644.931	105.6170	1110.230	61.3682
8192	537561.435	2394.450	224.5000	2719.380	197.5926

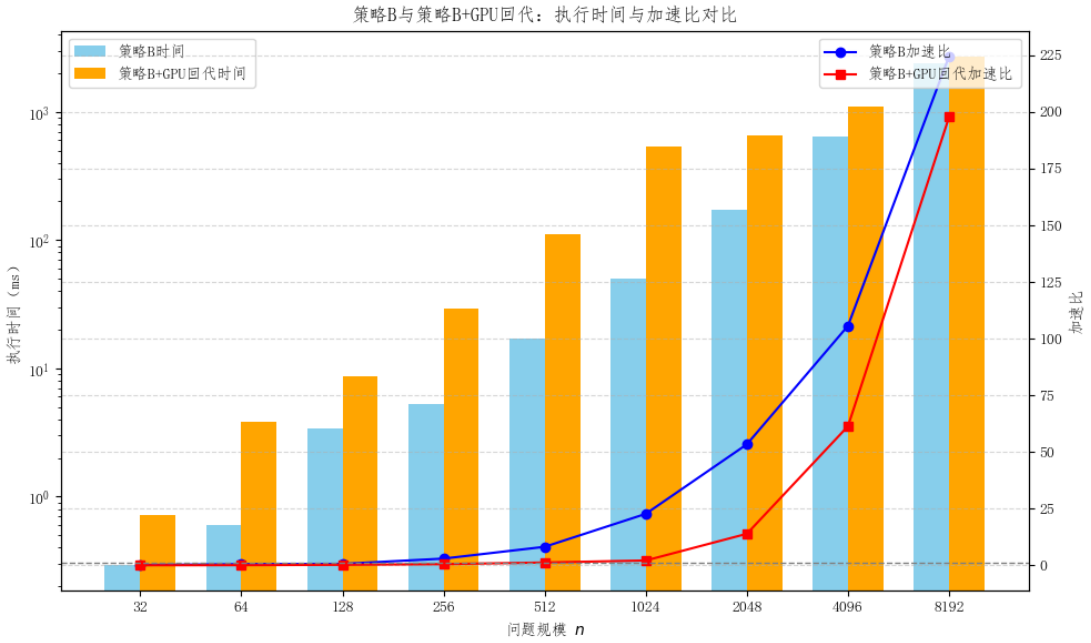

图 19: 策略 B 与策略 B+GPU 回代执行时间与加速比对比

尽管策略 B+GPU 回代在设计上实现了计算全过程的设备端并行，但实验结果显示，其整体性能反而低于只对消元阶段进行 GPU 加速的策略 B，尤其在中等规模问题上表现尤为明显。通过对折线图与表中数据的综合分析，我可以从多个层面剖析这一现象的原因：

- **回代阶段本身计算量较小，GPU 加速性差：**高斯消元中的回代过程具有明显的串行依赖特性，每个未知数的求解依赖于前面所有变量的值。因此其可并行性较差，计算粒度小。在中小规模问题上，使用 GPU 进行加速反而由于核函数启动开销、线程调度成本和设备同步延迟，导致整体性能下降。
- **线程资源利用率低，SM 并行度未充分释放：**回代阶段使用的 GPU 线程远少于消元阶段。例如，在 $n = 1024$ 时，回代所需的线程数最多为 n，远远小于消元阶段的 $n(n-1)$ 级别的线程需求，导致大量 SM 资源处于空闲状态，GPU 并未真正“满负荷”工作，反而增加了上下文切换和同步成本。
- **访存模式和内存带宽使用效率下降：**回代阶段主要为逐行读取与写入，线程之间缺乏良好的内存合并访问特性。这意味着回代核函数的全局内存访问带宽未得到高效利用，访存延迟成为瓶颈，从而进一步削弱了 GPU 的理论性能优势。
- **核函数调用频繁，调用开销被放大：**回代过程中的每步操作虽然在 GPU 中可通过线程并行处理每一项，但整个过程涉及递减式的迭代循环，每次需要重新调用核函数并同步，导致 CUDA 函数调用频繁，累积的启动开销难以忽略，特别是在问题规模 $n = 256$ 或 $n = 512$ 这一现象尤为显著。
- **回代阶段的优化空间有限：**尽管尝试了如共享内存加速、___restrict___ 和 const 修饰符提示、线程间分工等优化策略，但由于回代阶段数据依赖强、计算密度低的特点，这些优化手段带来的性能提升非常有限。

从实验数据中可见，在 $n \geq 4096$ 时，策略 B+GPU 回代才开始在加速比上逐渐接近策略 B，这说明只有在极大规模的矩阵问题中，回代所占的比例足够高，GPU 回代才能体现一定优

势。而在大多数实际场景下，GPU 回代所引入的额外开销反而成为性能瓶颈。

虽然完全 GPU 并行化具有理论上的完整性，但在实践中需权衡各阶段的计算密度与可并行性。回代阶段更适合保留在主机端串行执行，而将主要的计算密集型阶段移至 GPU，这种“混合并行”策略在性能与可扩展性之间达到了更优平衡。

(四) GPU 优化策略综合分析总结

GPU 高斯消去的性能优化涵盖了线程划分、内存管理、核函数设计与计算调度等多个维度。GPU 编程实验基于 CUDA 平台，系统地探索了不同优化策略对计算效率的影响，总结如下：

首先，在任务划分方面，**策略 B**（每个 block 负责一行，每个线程负责该行的一个列元素）凭借其细粒度的并行性，实现了 $O(n^2)$ 级别的线程并发，显著优于 **策略 A** 中一线程处理一整行所能提供的 $O(n)$ 并行度。策略 B 不仅提升了 SM 的占用率，还天然支持内存合并访问（memory coalescing），极大提高了全局内存带宽利用率，避免了线程空转与访存瓶颈。此外，通过线程协作、共享内存缓存主元行和合理同步，策略 B 能在保持正确性的同时大幅提升吞吐量，是当前实验中最优的任务划分方式。

其次,在 **内存优化策略**方面,实验尝试了共享内存、寄存器缓存与语义修饰符(如 `__restrict__` 和 `const`）等技术以减少冗余访存和提升编译器优化空间。但结果显示，在实际运行中，这些优化策略的性能提升有限。原因主要在于：共享内存带来的同步开销与资源竞争在中大规模问题中反而拖慢了执行效率，尤其在宽矩阵上频繁发生 bank conflict；而寄存器与编译器优化的效果也因指令复杂度和线程调度负担而被稀释。因此，**优化收益需综合考虑同步代价和内存容量瓶颈**，并非所有“看似合理”的优化都能提升整体性能。

在 **线程块配置**方面，实验结果清晰表明：较小的 block size（尤其是 64）在资源调度和并行度之间达到了理想平衡，适合高斯消元这类结构规律明确、计算密集型的任务。block size 越大，SM 并发调度能力越受限，尤其在 1024 的情况下，由于共享内存和寄存器资源紧张，反而导致性能显著下降。因此，block size 的选取应结合硬件资源限制和问题规模动态调优。

最后，在 **计算流程层级**方面，实验还进一步尝试将回代阶段从主机端迁移至设备端实现，实现全流程 GPU 并行化。还采用了 `atomicAddDouble` 实现 double 类型并发累加等手段优化回代性能。然而，由于回代过程串行依赖性强、并行粒度小、内存访问不规整，其在 GPU 上的执行不仅未提升性能，反而因核函数启动与同步开销造成额外负担。

综合而言，GPU 并行优化策略的成效在于：

- 提高并行粒度与线程活跃度。
- 优化访存模式以充分利用带宽。
- 减少核函数调度与同步带来的系统开销。
- 在“混合并行”架构中合理分配 CPU 与 GPU 的任务边界。

九、 新增内容

本次综合实验在以往阶段性作业基础上进行了统一规划、整合与改进，具体新增内容包括：

- **统一的三段式并行框架**：构建了通用于 SIMD、多线程、MPI、GPU 架构的统一算法设计模板，分别从主元标准化、消去过程、回代求解三个阶段进行结构化并行划分，便于对不同平台的实现策略进行横向对比。

- **算法设计分析**：架构间的共通与差异。其中 SIMD 的数据进行了更新（开启了-O2 优化）、多线程的数据进行了部分更新、MPI 的数据进行了部分更新。
- **多线程并行效率评估**：新增并行效率计算与分析内容，评估 Pthread 与 OpenMP 不同线程数下的加速比与资源利用效率，深入探讨线程数与问题规模之间的适配关系。
- **GPU 实验的完整并行化**：在原 GPU 实验基础上，完成了从矩阵初始化、显存拷贝、核函数设计到结果回传与回代的全流程并行化实现，并且与原实验进行对比。
- **其余内容总结与整合**：在保留阶段性作业核心思路的基础上，对各平台实验结果、代码优化策略与瓶颈问题进行了统一提炼与归纳。

十、 总结

经过这学期的并行程序设计实验，我对并行计算有了从“听说”到“动手实践”的全面认知，系统地掌握了多种并行计算平台的编程模型与优化策略。从最初接触 SIMD 指令集优化，到尝试用 Pthread 和 OpenMP 写多线程程序，再到使用 MPI 进行进程间通信，最后走向 CUDA 编程、调试 GPU 上的大规模矩阵计算，每一次实验都让我感受到不同并行平台背后的设计哲学和性能潜力。

在实现高斯消元的多个版本过程中，我体会到并行程序不仅仅是“代码写快一点”，而是需要对架构特性、任务划分、同步机制等方面进行全方位的考虑。很多时候，性能提升并不是来自“硬刚”和“堆叠”，而是来自巧妙的设计。

虽然并行程序设计这门课程实验多、内容广、思维跨度大，但收获也非常丰富。本课程极大地拓展了我对现代计算架构与高性能编程的认识，我学会了在并行计算中“因地制宜”地选择工具和策略，也更加理解了“优化不是堆代码，而是设计”的真正含义。除此之外，写好一份实验报告和写好一份程序一样重要，清晰地表达思路本身就是一种能力。相信这些经验会在我以后的学习生活中发挥长远的作用。

十分感谢老师的讲解和指导，感谢助教们的帮助。

源代码链接Github