南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计

SIMD 编程——高斯消去算法

孟启轩

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2025 年 4 月 28 日

摘要

结合课堂内容以及实验指导书，我对高斯消元算法进行不同程度的优化，实现了 NEON 算法、AVX 优化算法、SSE 优化算法，分别采取不同的优化策略并测试耗时来对比分析性能。

关键字：SIMD 高斯消元 arm x86 AVX SSE NEON 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];
}
```

(三) 对比分析方向

x86 平台

- 串行算法内存不对齐与内存对齐进行对比
- 串行算法、AVX 优化、SSE 优化之间的对比
- AVX4 路优化算法，分别在内存不对齐与对齐条件下，消去过程优化、回代过程优化和整体优化进行对比
- SSE 和 SSE3 之间的对比
- SSEO0 优化和 O2 优化的对比
- 串行算法的 cache 优化

arm 平台

- 串行算法与 NEON 优化算法进行对比
- NEON 优化算法内存不对齐与内存对齐进行对比
- NEON 优化算法，优化不同部分进行对比

二、 SIMD 算法设计与实现

(一) AVX4 路向量化

通过利用 AVX 指令集提供的 SIMD 并行计算能力，对高斯消元法中的核心矩阵运算，如行消去、标量乘法、向量加减等进行优化，将多个数据元素打包到单个 AVX 寄存器中同时处理，从而减少循环迭代次数、提升计算并行性，最终加速高斯消元过程。

1. 不对齐

对串行算法的关键循环部分进行 AVX4 路向量化优化，对于消去过程使用 AVX 的 256 位寄存器（___m256d）处理 4 个双精度浮点数，将内层循环的步长设为 4，以匹配 SIMD 宽度（j += 4），减少了循环迭代次数，提高了数据并行性。对于回代过程，使用向量化计算 sum += A[i][j] * x[j]，步长为 4，减少了标量循环次数。

虽然消去过程的时间复杂度仍为 $O(n^3)$,但减少了常数因子,回代过程时间复杂度仍为 $O(n^2)$,但减少了循环次数，性能得以提升。

AVX 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 * n + k] / A[k * n + k];
        __m256d factor_vec = _mm256_set1_pd(factor);

        int j = k + 1;
        for (; j + 3 < n; j += 4) {
            __m256d row_i = _mm256_loadu_pd(&A[i * n + j]);
            __m256d row_k = _mm256_loadu_pd(&A[k * n + j]);
            __m256d result = _mm256_sub_pd(row_i, _mm256_mul_pd(factor_vec,
                row_k));
            _mm256_storeu_pd(&A[i * n + j], result);
        }
        // 标量处理剩余元素...
    }
}
//回代过程
x[n - 1] = b[n - 1] / A[(n - 1) * n + (n - 1)];
for (int i = n - 2; i >= 0; i--) {
    int j = i + 1;
    __m256d sum_vec = _mm256_setzero_pd();
    double sum = 0.0;
    for (; j + 3 < n; j += 4) {
        __m256d a_vec = _mm256_loadu_pd(&A[i * n + j]);
        __m256d x_vec = _mm256_loadu_pd(&x[j]);
        sum_vec = _mm256_add_pd(sum_vec, _mm256_mul_pd(a_vec, x_vec));
    }
    // 将向量结果累加到标量sum
    // 处理剩余标量元素
    x[i] = (b[i] - sum) / A[i * n + i];
}
```


2. 对齐

内存对齐与不对齐大同小异，使用 _aligned_malloc 确保内存地址对齐到 32 字节，使用对齐指令，如 _mm256_load_pd 和 _mm256_store_pd，避免因未对齐导致 CPU 缓存重组等额外开销。

(二) SSE

利用 SSE 指令集对高斯消元法进行优化，通过 SIMD 并行计算提升性能。在消去和回代阶段，使用 SSE 的 128 位向量寄存器同时处理多个浮点数，将循环中的标量运算转化为向量运算（如 _mm_mul_ps、_mm_sub_ps），减少循环迭代次数，实现高斯消元中大量浮点计算的高效并行化。

对于消去过程，通过行变换将矩阵 A 转换为上三角矩阵。使用 ___m128 向量寄存器处理列

j 的循环，每次处理 4 个元素。外层循环 k 和 i 遍历矩阵行，内层循环 j 以步长 4 跳跃，减少标量循环次数。

对于回代过程，从最后一行开始，逐步求解方程组得到解向量 x。使用 SSE 向量指令计算 sum，减少标量循环的乘法和加法操作。剩余元素用标量循环完成。

虽然时间复杂度仍为 $O(n^3)$，但减少了常数因子，性能得以提升。

SSE 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) {
        float factor = A[i * n + k] / A[k * n + k];
        __m128 factor_vec = _mm_set1_ps(factor);  // 广播因子到向量
        int j = k + 1;
        // 向量化处理: 每次处理4个元素
        for (; j + 3 < n; j += 4) {
            __m128 a_i = _mm_loadu_ps(&A[i * n + j]);// 加载当前行的4个元素
            __m128 a_k = _mm_loadu_ps(&A[k * n + j]);// 加载基准行的4个元素
            __m128 res = _mm_sub_ps(a_i, _mm_mul_ps(factor_vec, a_k));// 向量
                运算: a_i -= factor * a_k
            _mm_storeu_ps(&A[i * n + j], res);// 存储结果
        }
        for (; j < n; ++j)// 标量处理剩余元素（不足4个时）
            A[i * n + j] -= factor * A[k * n + j];
        // 更新右侧向量b
        b[i] -= factor * b[k];
    }
}
//回代过程
for (int i = n - 1; i >= 0; --i) {
    float sum = b[i];
    int j = i + 1;
    __m128 sum_vec = _mm_setzero_ps();  // 向量累加器初始化
    // 向量化处理: 每次处理4个元素
    for (; j + 3 < n; j += 4) {
        __m128 a_vec = _mm_loadu_ps(&A[i * n + j]);  // 加载当前行的系数
        __m128 x_vec = _mm_loadu_ps(&x[j]);          // 加载解向量的4个元素
        sum_vec = _mm_add_ps(sum_vec, _mm_mul_ps(a_vec, x_vec));  // 向量累加
            a_j * x_j
    }
    // 将向量结果转换为标量
    float temp[4];
    _mm_storeu_ps(temp, sum_vec);
    sum -= temp[0] + temp[1] + temp[2] + temp[3];  // 合并向量结果
    for (; j < n; ++j)// 标量处理剩余元素
        sum -= A[i * n + j] * x[j];

    x[i] = sum / A[i * n + i];  // 计算当前解
}
```

SSE 不同版本优化

SSE 主要依赖基础 SIMD 指令实现向量化，而 SSE3 主要从以下方面进行优化：

- **数据预取优化：** 在循环中插入 _mm_prefetch 预取后续数据，减少内存访问延迟，缓解缓存缺失问题。
- **水平加法指令：** 回代阶段使用 SSE3 的 _mm_hadd_ps 水平加法指令，将向量元素逐级合并（先合并为 2 元素，再合并为 1 元素），直接提取标量结果，避免手动拆分和标量求和的开销。
- **指令流水线优化：** 通过水平加法指令减少中间步骤，提升向量到标量转换的效率，充分利用 SSE3 的指令级并行性。

(三) NEON

NEON 算法是基于 ARM 架构的 NEON SIMD 指令集对高斯消元法进行的优化实现，其核心是利用 NEON 的单指令多数据特性，通过向量化浮点运算加速消去阶段和回代阶段的计算，同时通过数据对齐和循环展开减少内存访问延迟，最终提升线性方程组求解的效率，尤其适用于大规模矩阵运算。

对于消去过程，使用 vdupq_n_f32 将标量因子扩展为 4 元素向量。通过 vld1q_f32 加载 4 个元素，执行 vmulq_f32 和 vsubq_f32，并用 vst1q_f32 存储结果。j 每次递增 4，处理 4 个元素，减少循环次数。

对于回代过程，使用 vmlaq_f32（sum_vec += a_vec * x_vec）合并乘法和加法，减少指令数。

虽然时间复杂度仍为 $O(n^3)$，但减少了常数因子，性能得以提升。

NEON 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) {
        float factor = A[i * n + k] / A[k * n + k];
        float32x4_t factor_vec = vdupq_n_f32(factor); // 向量扩展因子
        int j = k + 1;
        for (; j + 3 < n; j += 4) { // 处理4个元素
            float32x4_t a_i = vld1q_f32(&A[i * n + j]);// 加载当前行向量
            float32x4_t a_k = vld1q_f32(&A[k * n + j]);// 加载主元行向量
            float32x4_t res = vsubq_f32(a_i, vmulq_f32(factor_vec, a_k));//
                向量计算
            vst1q_f32(&A[i * n + j], res);// 存储结果
        }
        for (; j < n; ++j)
            A[i * n + j] -= factor * A[k * n + j];
        b[i] -= factor * b[k];// 更新常数项
    }
}
// 回代过程
for (int i = n - 1; i >= 0; --i) {
    float sum = b[i];
```

```
    int j = i + 1;
    float32x4_t sum_vec = vmovq_n_f32(0.0f);// 初始化向量
    for (; j + 3 < n; j += 4) { // 处理4个元素
        float32x4_t a_vec = vld1q_f32(&A[i * n + j]);// 加载系数
        float32x4_t x_vec = vld1q_f32(&x[j]);// 加载已知解
        sum_vec = vmlaq_f32(sum_vec, a_vec, x_vec);// 向量乘加
    }
    // 向量合并为标量
    float32x2_t sum_low = vget_low_f32(sum_vec);// 取低2元素
    float32x2_t sum_high = vget_high_f32(sum_vec);// 取高2元素
    float32x2_t total = vpadd_f32(sum_low, sum_high);// 合并为2元素
    float final_sum = vget_lane_f32(vpadd_f32(total, total), 0);// 合并为标量
    sum -= final_sum;
    for (; j < n; ++j)
        sum -= A[i * n + j] * x[j];
    x[i] = sum / A[i * n + i];
}
```

三、 实验结果分析

通过不同问题规模下的耗时进行不同优化策略的对比分析，问题规模为 2 的整数次方。

(一) x86 平台下的实验结果分析

1. 串行算法内存不对齐与对齐对比

串行高斯消元算法在 x86 架构下，内存对齐优化对性能具有一定的影响。在小规模矩阵中，对齐与不对齐的性能差异较小；但随着问题规模增长，内存对齐版本的执行效率会有小幅度提升。内存对齐通过减少缓存未命中和提升数据访问连续性，能有效降低性能损耗，尤其在处理大规模矩阵时优势更为突出。

表 1: 串行高斯消元算法内存不对齐与对齐耗时对比（单位：微秒）

问题规模	不对齐	对齐
32	32	40
64	246	241
128	1966	1923
256	15 284	14 765
512	123 336	120 413
1024	987 865	948 374
2048	8 015 784	7 882 569

汇编分析

使用 Godbolt 对代码进行汇编层面的分析。

发现在不对齐串行代码里，很多地方是这样的模式：

```
lea     rdx, [0+rax*8]
mov     rax, QWORD PTR [rbp-56]
add     rax, rdx
mov     rax, QWORD PTR [rax]
```

主要数据操作是 movsd，地址访问是先算偏移 rax + rdx，然后再取 [rax]，没有特殊保证对齐。每次都是 movsd、mulsd、divsd 这样的标量指令在处理，处理的数据源也是不对齐的，尽管是标量，但 load 延迟已经打慢了流水线。

当访问地址不是内存对齐时，CPU 必须做跨缓存行的读写，会需要两次加载或者需要硬件额外合并。此外，不对齐的数据访问更容易触发乱序窗口中的加载违规 load violation，导致流水线重新执行，stall 次数增加。并且写内存时，缓存行合并对齐时更高效，不对齐时需要额外处理部分写入。

而内存对齐代码通过 _aligned_malloc，在汇编里用 movapd，加载的时候就可以一次对齐拿到数据，减少访存 stall，尤其在消去过程这种长循环中，累计效果很明显。

内存对齐能显著减少 CPU 访存次数、降低指令开销、避免访存 stall，从而大幅提升串行高斯消元这种访存密集型程序的整体性能。

2. 串行算法、AVX 优化、SSE 优化之间的对比

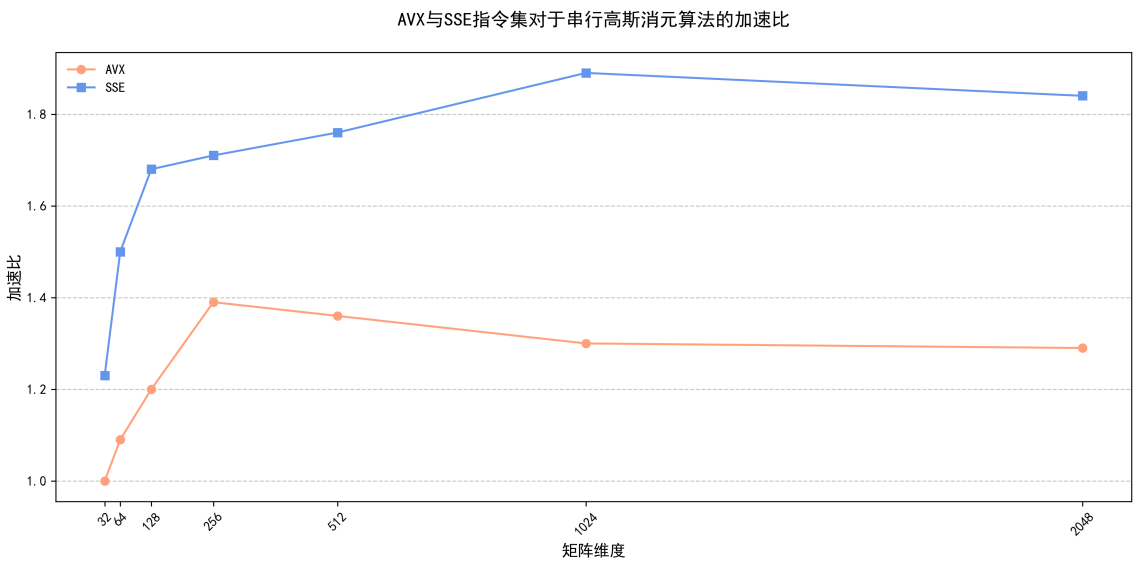

图 1: AVX 与 SSE 指令集对于串行高斯消元算法的加速比

内存不对齐条件下，对比不同规模下串行算法、AVX 优化算法、SSE 优化算法的耗时，以及 AVX 和 SSE 对于串行的加速比，分析得到：随着问题规模的增大，串行、AVX 和 SSE 三种方法的计算耗时均显著增加，但 AVX 和 SSE 相比串行算法均表现出加速优势，且加速比随问题规模扩大而呈上升趋势。具体来说，AVX 的加速比从 32 阶的 1.00 增长至 2048 阶的 1.29，而 SSE 的加速比则从 1.23 显著提升至 1.84，可以看出 SSE 在大规模数据处理中具有更明显的性能优势。这说明在高斯消元算法中采用 AVX 和 SSE 指令集能有效提升计算效率，SSE 的优化效果要更好一些，并且在处理大规模矩阵时，SSE 的优化效果更为突出。

虽然理论上 AVX 指令集由于单指令能处理 256 位数据，相较于 SSE 的 128 位应当具备更高的并行度和更优的性能，但在本次高斯消元算法的实际测试中，SSE 优化反而表现出更好的加

速比。可能是因为高斯消元属于强数据依赖型算法，存在大量的逐步依赖关系，使得指令流水线难以充分发挥 AVX 的宽向量优势，反而因较大寄存器操作和调度开销拖慢了执行速度。

3. AVX4 路优化算法深度剖析

对于 AVX4 路优化算法，分别在内存不对齐与对齐条件下，对消去过程优化、回代过程优化和整体优化进行对比分析。

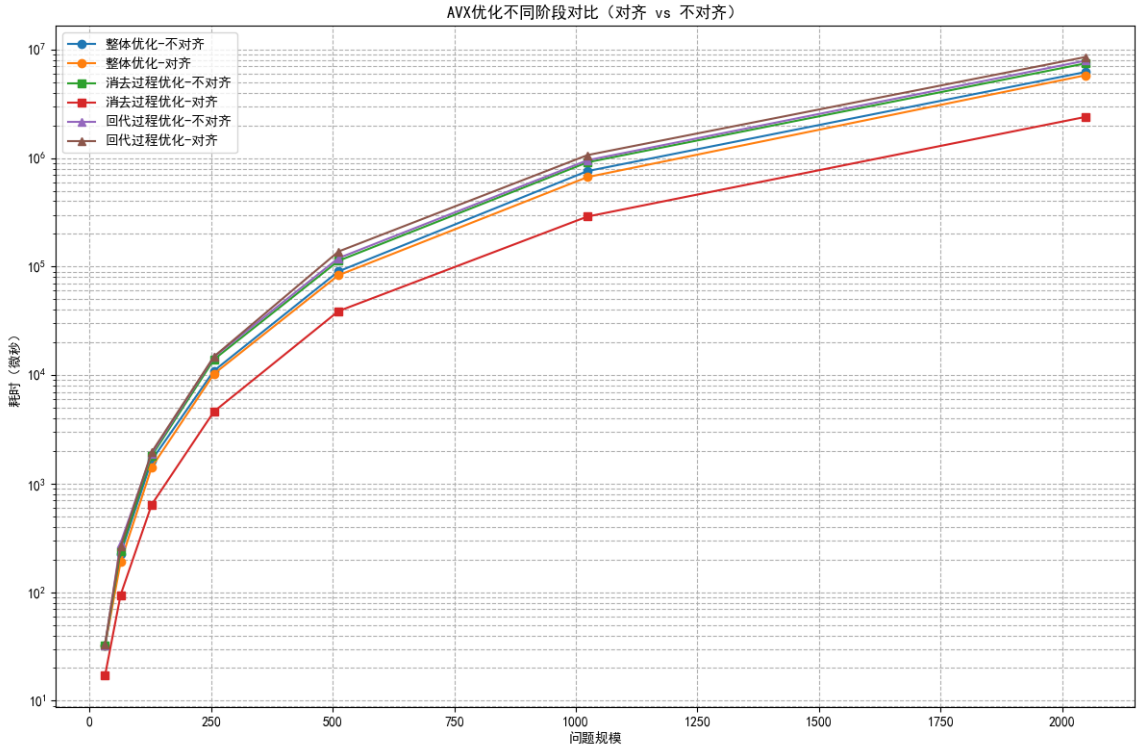

图 2: AVX 优化不同阶段对比

首先，在整体 AVX 优化的对比中，内存对齐条件下的耗时始终小于不对齐条件下，且随着问题规模的扩大，对齐带来的性能优势逐渐扩大。只对消去过程优化时，可以看到内存对齐使得计算速度大幅提升。这种加速主要得益于 AVX 能够高效加载对齐内存块，减少了内存访问延迟。而在回代过程优化中，低阶规模对齐与否影响不大，但在高阶规模对齐后甚至比不对齐性能反而下降，出现了异常现象。推测原因是回代过程数据访问模式存在大量非连续访问，导致对齐反而引发了额外的 cache miss 和无效加载，影响了整体性能。

对于高斯消元算法，主要的时间复杂度在消去过程，所以优化消去过程对于性能的提升十分显著。一般来说，整体优化要比单独优化某一部分的效果要好很多，但由于 AVX 对于回代过程的优化在内存对齐的情况下，问题规模较大时性能反而下降，所以导致整体优化的性能不如只优化消去过程的性能。

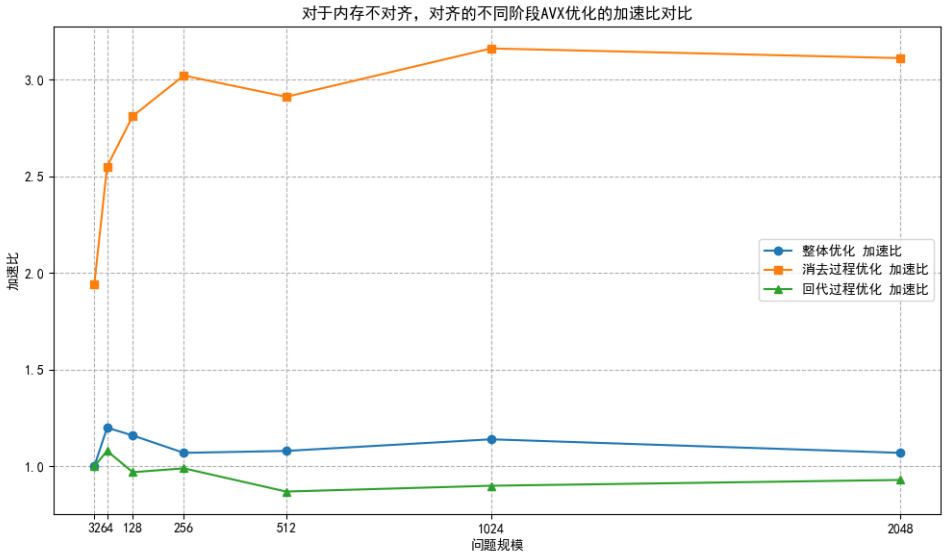

图 3: 对于内存不对齐，对齐的不同阶段 AVX 优化的加速比对比

通过加速比对比分析可以发现，在整体优化中，数据对齐带来的性能提升有限，加速比基本稳定在 1.1 左右，说明整体流程中仍存在较多受限于内存带宽或指令调度的瓶颈。而在消去过程优化中，对齐优化的效果极为明显，加速比稳定在 2.5 至 3 倍以上，说明消去阶段的向量化计算密集、内存访问频繁，且高度依赖数据对齐特性，因此在这一阶段数据对齐可以显著减少 CPU 访问延迟，充分释放 AVX 指令的并行潜力。相比之下，回代过程的加速比则整体低于 1，甚至存在对齐后性能略微下降的情况，推测可能是由于回代阶段计算强度较低、数据访问模式不规则，导致对齐优化带来的内存访问加速不足以弥补额外的指令开销，也有可能是串行处理剩余数据的原因。综上所述，对消去阶段进行对齐优化，可以最大化发挥 AVX SIMD 指令集的性能优势。

fma

我还进行了 AVX 的 fma 优化，消去过程使用 _mm256_fnmadd_pd 替代 mul+sub 指令，将乘减操作合并为单个指令，回代过程使用 _mm256_fmadd_pd 替代 mul+add 指令，实现乘加融合，相对于 AVX 优化加速比约为 1.3。

4. SSE 和 SSE3 之间的对比

SSE 优化表现更为优越，特别是在消去阶段的向量化优化上。SSE3 的引入虽然在回代阶段通过水平加法 _mm_hadd_ps 略微提升了性能，但由于小矩阵中 _mm_prefetch 的引入反而增加了内存带宽压力，导致 SSE3 在较小矩阵时反而比 SSE 慢。对于大矩阵，SSE3 的预取机制并未能有效提升性能，并且由于回代阶段的水平加法指令引入了额外的延迟，SSE 的简洁指令流和直接的内存访问模式让其整体表现更好。

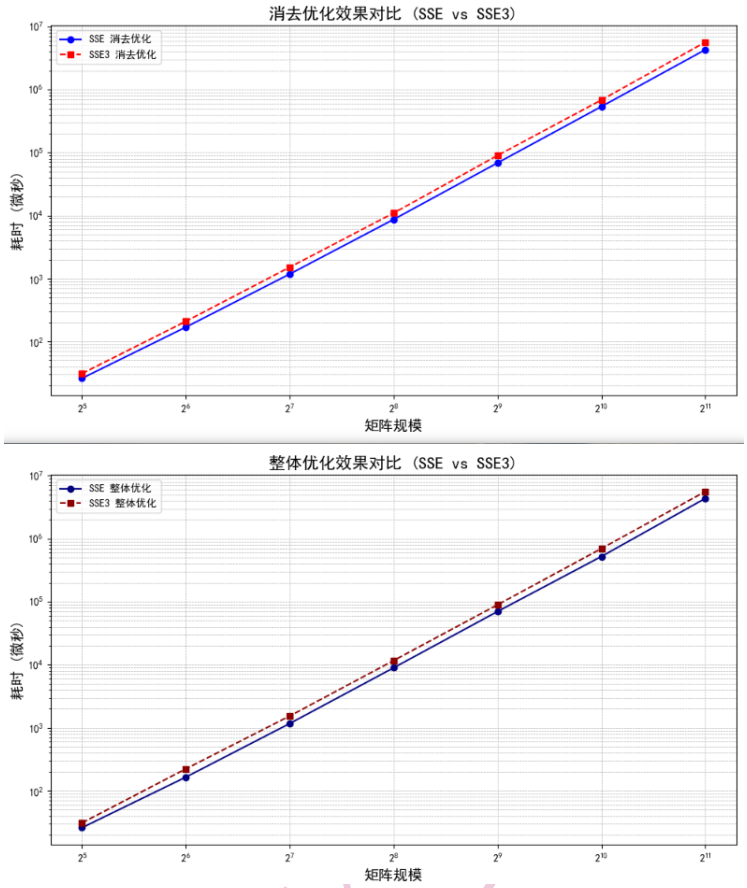

图 4: SSE 与 SSE3 对比

5. SSEO0 优化和 O2 优化的对比

分别对 SSE 和 SSE3 进行-O0 和-O2 优化，进行对比分析。

-O0

- 禁用所有优化，编译器生成最直白、最保守的代码。
- 不能很好利用寄存器，数据会反复写回内存。
- 不能指令重排，流水线效率差。
- 不能内联函数，循环展开，不能消除冗余计算。
- SIMD 指令虽然写了，但生成的机器码也没很好利用。

-O2

- 自动使用更多寄存器。
- 自动展开小循环。
- 常量传播、死代码消除、强烈调度 SIMD 指令。
- 充分利用了 SSE、SSE3 指令集潜力。

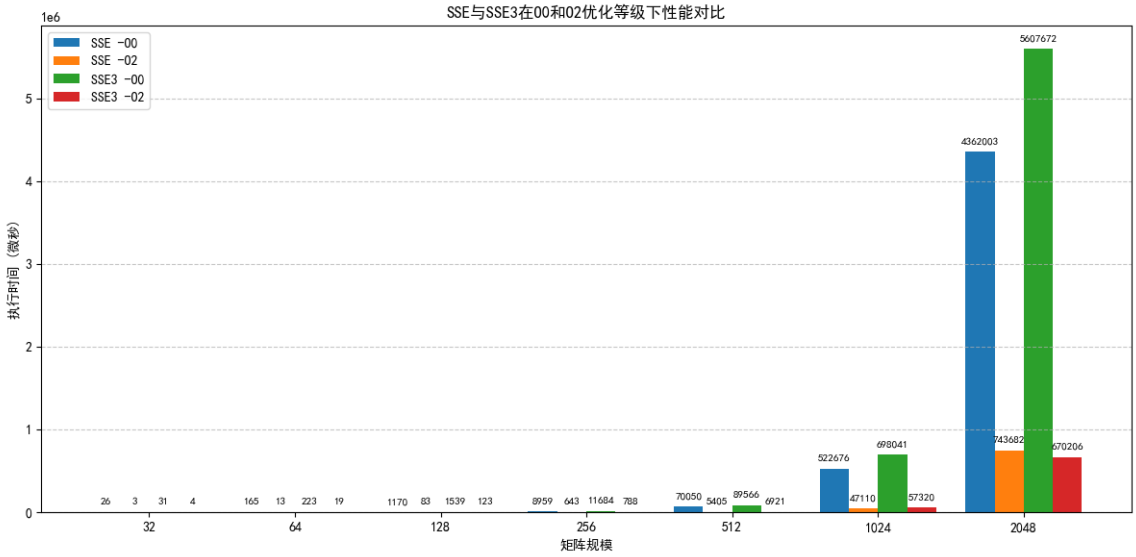

图 5: SSE 与 SSE3 在 O0 和 O2 优化等级下性能对比

小规模时优化幅度没那么明显，矩阵较大时，-O2 优化优势成倍放大，因为循环更多，访存更密集，优化带来的累计节省越来越大。无论是 SSE 还是 SSE3，-O2 优化能帮助实现真正的 SIMD 并行化，使算法性能得到极大的提升。尤其是 SSE3 -O0 在 2048 规模时接近 5607672 µs，而 SSE3 -O2 只有 670206 µs，将近 8.54 倍加速，这个优化力度是非常可观的。

所以即使用 SSE、SSE3，如果不开优化，性能也很差，必须有-O2 帮助，才能发挥 SIMD 的真正力量。

6. 串行算法的 cache 优化

二维数组虽然内存上是连续的，但行是独立分配的，new 了很多小块。动态分配的小块内存，很容易导致内存碎片化，降低了预取器、缓存行的效果。另外，访问数据时，跨行访问导致 cache miss 多，局部性差。而且每次 A[i][j] -= factor * A[k][j]，其实是在遍历一整行，非常依赖行连续。我在大部分程序中使用一维数组来代替二维数组，这样内存布局紧凑，可以线性访问数据。

此外，我还进行了以下 cache 优化，加速比在 1.2 左右。

- **循环优化：** 消去过程将 j 循环展开 4 路，增加指令级并行。并调整内存访问模式，确保最内层循环访问连续内存地址。还对行首地址进行预计算，减少重复计算。
- **缓存友好访问：** 消去过程中，同一行的元素按顺序访问，充分利用缓存行。回代过程采用合并访问模式，减少缓存抖动。

串行算法 cache 优化核心代码

```
// 消去过程
for (; j <= n - 4; j += 4)
{ // 4路循环展开
    A[row + j] -= factor * A[pivot + j];
    A[row + j + 1] -= factor * A[pivot + j + 1];
    A[row + j + 2] -= factor * A[pivot + j + 2];
    A[row + j + 3] -= factor * A[pivot + j + 3];
}
```

```
// 回代过程
for (; j <= n - 4; j += 4)
{ // 4路循环展开
    sum -= A[row + j] * x[j] + A[row + j + 1] * x[j + 1] + A[row + j + 2] * x
        [j + 2] + A[row + j + 3] * x[j + 3];
}
```

(二) arm 平台下的实验结果分析

1. 串行算法与 NEON 优化算法进行对比

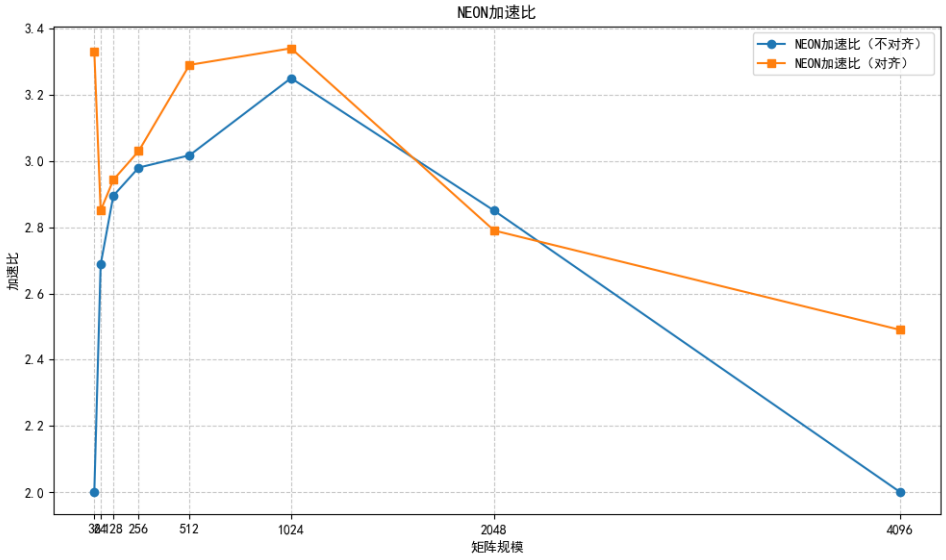

图 6: NEON 加速比

- **串行 vs NEON 整体优化：**不管是对齐还是不对齐，NEON 整体优化的执行时间都远小于串行。特别是规模大的时候，比如问题规模为 4096 时，NEON 整体优化比串行快了 2 倍甚至更多。NEON 并行加速非常显著，大矩阵时十分明显。
- **不对齐 vs 对齐：**对齐数据在大矩阵时明显比不对齐快。小矩阵时 NEON 差别很小，基本无影响，而串行算法反而出现了性能下降的问题。

在 ARM 架构下, NEON 向量化算法与串行算法的性能差异在数据规模和访存特性上呈现显著分别。对于大规模问题, NEON 指令的性能优势依赖于严格的内存对齐。当内存对齐时, NEON 可通过加载/存储对齐指令一次性完成 128 位数据的高效搬运，减少缓存行的跨边界访问，避免因未对齐导致的多次分段加载或硬件合并开销。这种对齐优化能显著降低访存延迟，提升 SIMD 指令的吞吐率；而未对齐访问则会触发额外的内存操作，导致流水线停顿和乱序执行的异常处理，从而抵消向量化的加速效果。相比之下，串行算法因为依赖标量指令，其访存模式本身对齐需求较低，并且计算密集度相对较低，因此即使未对齐的访存开销在整体执行时间中占比有限，但整体性能仍受限于标量运算的效率。因此，在 ARM 平台中，内存对齐的 NEON 向量化优化能充分释放 SIMD 的潜力，而串行算法虽然受访存影响较小，受限于标量计算效率，内存对齐与否性能差别不大。

2. NEON 优化算法内存不对齐与内存对齐进行对比

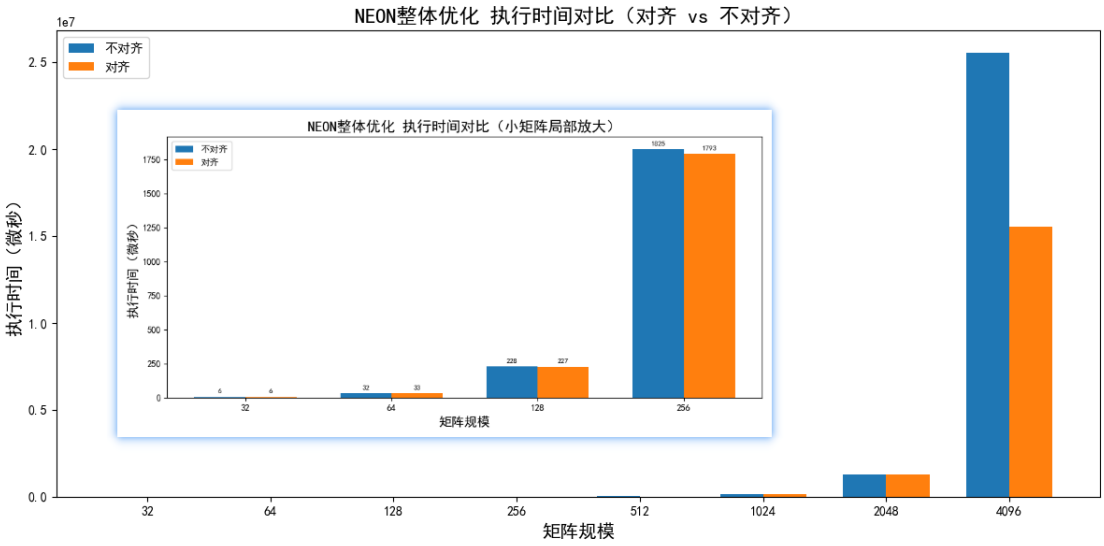

图 7: NEON 内存不对齐与内存对齐耗时对比

对于 NEON 算法，内存对齐与否对整体性能影响不大，只有在很大规模的矩阵时，内存对齐的性能优势比较明显。分析可能有以下几点原因:

- **NEON 的乱序和容错机制：**ARM NEON 单元硬件支持一定程度的乱序访问，能够自动处理轻微的未对齐访问。因此，在小矩阵中，内存对齐与否的性能差距非常小。
- **小矩阵的缓存命中率较高：**小规模矩阵的数据大部分可以保存在 L1/L2 缓存中，避免访问较慢的主内存。而鲲鹏服务器的内存访问和缓存架构非常高效，缓存命中率极高，这使得对齐和不对齐对性能的影响不明显。
- **大矩阵时内存带宽成为瓶颈：**对于大规模矩阵（如 4096×4096），数据量超出 L1/L2 缓存的容量，必须频繁访问主内存。此时，未对齐的访问可能需要额外的拆分和重组，导致跨 Cache Line 的访问，增加了内存访问延迟和带宽浪费。而对齐的数据能实现高效的内存访问，减少访存延迟，显著提高性能。
- **编译器优化和 CPU 预取器的局限性：**尽管 -O2 优化能够提高循环展开和数据预取，但对于未对齐的数据，编译器和 CPU 预取器难以做出有效的优化。对齐的数据更容易被 CPU 预测并批量预取，减少了内存访问的等待时间。因此，随着矩阵规模的增大，未对齐带来的性能劣势会逐渐放大。

3. NEON 优化算法，优化不同部分进行对比

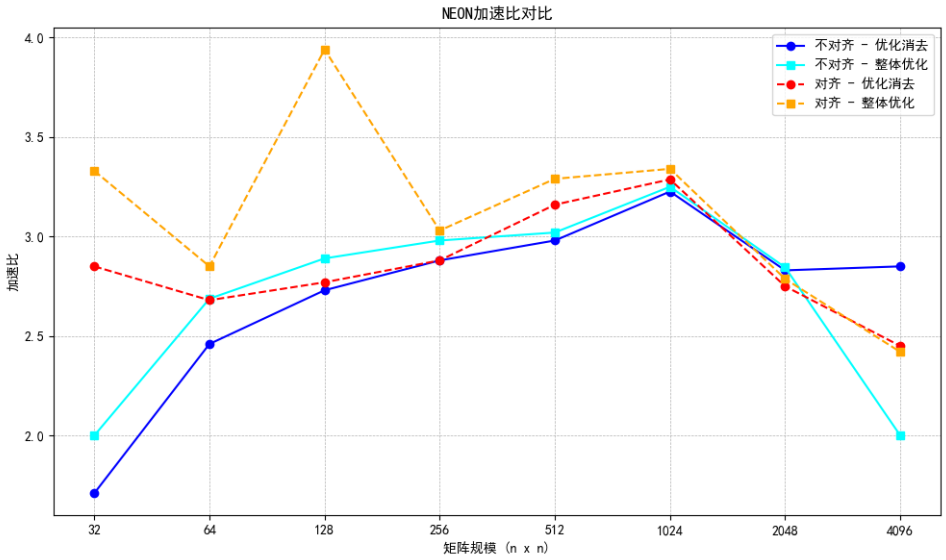

图 8: NEON 加速比对比

整体趋势同 x86 平台的 AVX 优化相同。整体加速比随着矩阵规模的增大，大体上比较稳定，大部分在 2.5 - 3.5 之间。优化消去阶段的加速比通常更高接近 3。而优化回代阶段的加速比较低，甚至出现了小于 1 或接近 1 的情况，回代过程优化不明显。

因为是密集的线性代数操作，消去过程可以很好地实现 SIMD 并行化，NEON 指令在这类密集运算中效率极高，能够充分并行执行多条指令。对齐可以让访存效率更好，但因为消去过程中重用数据较多，对齐带来的提升并不特别大。而回代操作每步依赖前一步的结果，数据相关性强。NEON 虽然可以并行处理，但在高度数据依赖场景下并行度下降。特别在大规模矩阵下，访存压力大、依赖链长，导致回代阶段很难提速，甚至因为数据组织、加载等额外开销导致略微变慢，无法充分发挥 SIMD 并行优势。

四、 遇到的问题以及说明

- AVX-512 优化算法未能正确实现，尝试修改多次过后，程序仍然运行失败并且没有定位到问题，推测可能是和我实现的内存对齐有冲突。
- 只要没有单独说明，在 x86 平台的实验中，我并没有在编译时开启优化，这样可能没有完全发挥出 SIMD 的优势。
- arm 架构下的程序都是开启-O2 优化进行对比的。

五、 总结

通过本次实验，我深入理解了高斯消元算法的性能瓶颈及优化思路，初步学习使用了鲲鹏服务器，对 arm 平台下的程序测试与分析也有了初步的了解。实验通过不同策略的优化与分析，不同程度上提升了算法的性能，加深了我对底层计算机体系结构与程序性能关系的理解，为今后进行更高效的并行优化打下了坚实基础。

源代码链接Github