南開大學
Nankai University

计算机学院

并行程序设计实验报告

高斯消去算法和 LU 算法的 SIMD 加速

姓名：蔡沅宏 黄明洲

学号：2213897 2211804

专业：计算机科学与技术

2024 年 4 月 28 日

目录

1 任务描述

1.1 研究任务及报告说明

本研究旨在分别对 gauss 方法和 LU 算法进行并行化优化，运用各种 simd 方法进行性能提升。通过实验数据分析，比较不同优化方法的性能表现，并根据实验结果对两种方法进行横向对比，分析它们适用的情况。

本报告主要展示小组成员蔡沅宏针对 LU 算法展开的一系列研究与分析，报告将在结尾模块展示小组成员完成各自任务后针对这两种算法应用的合作分析结果。

1.2 研究问题说明

1.2.1 Guass 消去

在进行科学计算的过程中，经常会遇到对于多元线性方程组的求解，而求解线性方程组的一种常用方法就是 Gauss 消去法。即通过一种自上而下，逐行消去的方法，将线性方程组的系数矩阵消去为主对角线元素均为 1 的上三角矩阵。Gauss 消去可用来进行线性方程组求解，矩阵求秩，以及逆矩阵的计算。

高斯消元法计算公式：

- 将方程组表示为增广矩阵 [A|b]，通过消元操作将系数矩阵 A 化为上三角矩阵；
- 利用回代求解出方程组的解 x；

Gauss 消去示意图如1.1所示：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1\ n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2\ n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1\ 1} & a_{n-1\ 2} & \cdots & a_{n-1\ n-1} & a_{n-1\ n} \\ a_{n\ 1} & a_{n\ 2} & \cdots & a_{n\ n-1} & a_{n\ n} \end{bmatrix} => \begin{bmatrix} 1 & a'_{12} & \cdots & a'_{1\ n-1} & a'_{1n} \\ 0 & 1 & \cdots & a'_{2\ n-1} & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & a'_{n-1\ n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

图 1.1: Gauss 消去示意图

1.2.2 LU 算法

LU 分解是一个重要的数值分析方法，它将矩阵分解为下三角矩阵 L 和上三角矩阵 U 。在科学计算、工程技术等领域中具有广泛应用。通过 LU 分解将原始线性方程组转化为两个简单的三角形方程组，从而可以更容易地求解线性方程组。同时可以用 L 和 U 的对角线元素相乘得到原矩阵的行列式。此外，通过 LU 分解后，可以快速求解矩阵的逆矩阵，从而可以方便地进行矩阵运算。

LU 分解算法计算公式：

- 将系数矩阵 A 分解为两个矩阵 L 和 U 的乘积，其中 L 为下三角矩阵，U 为上三角矩阵；
- 利用 LU 分解可以简化方程组求解的过程，提高计算效率；

LU 算法示意图如1.2所示：

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix}$$

图 1.2: LU 算法示意图

2 实验环境

此实验将对 LU 算法在 x86 和 ARM 两个平台下进行算法并行优化及分析，下表是对于实验环境详细展示，如表1所示：

	ARM	**x86**
CPU 型号	华为鲲鹏 920 服务器	12th Gen Intel Core i7-12700H
CPU 主频	2.6Ghz	2.3Ghz
一级缓存	64KB	1.2MB
二级缓存	512KB	11.5MB
三级缓存	48MB	24.0MB
指令集	NEON	SSE、AVX、AVX512

表 1: 实验环境

3 实验设计与分析

LU 算法中，在计算下三角矩阵 L 和上三角矩阵 U 的过程中，需要对矩阵的每一行进行向量乘法和加法操作，这些操作可以通过 SIMD 指令同时对多个元素进行并行计算，从而提高计算效率。另外，在 LU 分解的过程中，还涉及到一些对角线元素的除法运算，也可以通过 SIMD 指令实现并行计算，提高计算速度。

针对 LU 分解的过程（计算下三角矩阵 L 和上三角矩阵 U 的过程中），通过使用向量化指令，我们可以同时处理多个数据点在同一维度上的计算，从而加速 LU 分解算法的执行过程。例如在计算上三角矩阵时，我们使用多路向量操作来同时处理多个数据点的计算，用向量加载和向量乘法操作来加速计算。这样可以有效利用 CPU 的并行计算能力，提高计算速度和效率。当然，在转为 simd 并行化后，在每一轮计算中，都可能遇到多路向量无法刚好解决完所有数据元的问题，这时还需通过串行算法对余下的一些数据元进行操作，在提高程序的效率和性能的用时保证程序的正确性。

通过以上方式，我们可以对 LU 算法进行 SIMD 并行优化，提高计算速度和效率，从而更高效地进行矩阵分解操作。通过使用 SIMD 并行化技术，同时处理多个数据元素，提高 LU 算法的计算效率。在处理大规模矩阵时，采用 SIMD 并行化技术能够有效地加速 LU 分解的计算过程，实现更快速的线性方程组求解和矩阵运算。

本实验将在 x86、ARM 双平台上对算法的 simd 优化进行尝试和探究，并且将在串行、并行两方面对算法优化性能进行全面分析。

本实验涉及的所有代码都已上传 Github。

3.1 串行实现

首先，实现 LU 算法的串行版本，伪代码如下所示：

Algorithm 1 LU Decomposition

Initialize lower and upper matrices with zeros
Perform LU decomposition
for $i = 0$ to $n - 1$ **do**
    $lower[i][i] = 1.0$
    for $j = i$ to $n - 1$ **do**
        $sum = 0.0$
        for $k = 0$ to $i - 1$ **do**
            $sum+ = lower[i][k] \times upper[k][j]$
        end for
        $upper[i][j] = mat[i][j] - sum$
    end for
    for $j = i + 1$ to $n - 1$ **do**
        $sum = 0.0$
        for $k = 0$ to $i - 1$ **do**
            $sum+ = lower[j][k] \times upper[k][i]$
        end for
        $lower[j][i] = \frac{mat[j][i]-sum}{upper[i][i]}$
    end for
end for

3.2 x86 平台下

本实验中将算法迁移到了 x86 平台上，采用 x86 中的 SSE、AVX 和 AVX512 指令集架构分别对算法进行重构，然后对比实验效果。以下是本机支持的指令集，如表2所示：

指令集架构	版本
SSE	SSE/SSE2/SSE3/SSE4.1/SSE4.2
AVX	AVX/AVX2

表 2: CPU 支持指令集

为更深入研究优化比与不同 simd 方法的关系，除了尝试使用不同的指令集架构外，我还尝试了使用不同路向量进行优化，对于它们带来的不同优化比进行了分析。

同时为探究算法本身对并行问题带来的影响，我在 SSE 并行优化中，还尝试通过部分优化的方式（只优化计算上三角矩阵部分或只优化下三角矩阵部分）以探究算法本身可能对并行性能产生的影响，并且对一些现象进行了分析。

在并行化过程中也会遇到算法内存不对齐的情况，因此在此次实验中，还对 SSE 内存对齐与内存不对齐的算法的性能进行了分析。

3.3 ARM 平台下

本实验还在 ARM 架构下的鲲鹏服务器上完成了算法的 NEON 指令集的并行化优化测试，测试结果将与串行算法在鲲鹏服务器上的运行结果进行比较，然后分析现象以得到相应的实验结果。

由于在不同指令集架构下，对于部分优化与全部优化的区别问题、内存对齐问题、以及使用不同路向量优化效果不同的问题大同小异，这里就不再 ARM 平台上对这些问题进行讨论。

3.4 串行优化算法

除要求的对于代码进行并行优化外，我还在串行 LU 算法中探索其它优化方法，发现普通的 LU 算法将 L、U 矩阵分别进行循环计算，这可能导致额外的时间开销，所以我想尝试将两个过程合并一下，于是在尝试和调查过程发现类似思想的 LU 优化算法——Count_LU。因此，我还在实验中实现了 Count_LU 算法以探索串行优化的效果，并与并行优化进行了效果比较。

4 实验测试结果及分析

为方便测算与分析，我们在实验中使用的矩阵维度均为 2^n，以下报告中涉及维度时所述均为 n（表示是 2 的 n 次方）。

我们通过逐步改变矩阵大小，测算不同算法完成 LU 分解所需时间，来对不同算法的性能进行比较。同时，对不同规模下，相同算法的优化比的变化趋势进行分析；对相同规模下，不同算法的实际与理论优化比的不同进行分析，以进一步分析深层原因。

4.1 x86 平台下优化

x86 平台下的实验分析将从以下几个方面展开：

- 串行算法与 SSE2 路优化、4 路优化的比较
- 串行算法与 AVX4 路优化、8 路优化的比较
- SSE 与 AVX 方法的比较
- SSE 部分优化与全部优化效果的比较
- SSE 内存对齐与不对齐算法的效果比较
- 串行优化以及串行优化后并行优化探索

4.1.1 串行算法与 SSE2 路优化、4 路优化的比较

在 x86 平台下，通过测试，我们得到以下数据结果，如表3所示：

规模	x86 运行时间（ms）			规模	x86 运行时间（ms）		
	未优化	SSE			未优化	SSE	
		SSE2 路向量优化	SSE4 路向量优化			SSE2 路向量优化	SSE4 路向量优化
2	0.00734	0.00779	0.007482	7	10.2628	5.7357	3.1794
3	0.01438	0.01465	0.01448	8	73.5796	37.3459	19.0186
4	0.0437	0.03939	0.03344	9	572.377	306.647	145.698
5	0.19859	0.15439	0.10211	10	4668.5	2388.63	1149.67
6	1.27413	0.82146	0.48808	11	38710	19370.1	9269.96

表 3: 实验数据

我们发现，尽管我们分别采用了 2 路向量和 4 路向量对串行代码进行优化，但实际上得到的优化比，并没有办法达到 2 或 4，我们根据代码分析一下原因：

1. 在使用 SIMD 指令时，需要保证数据对齐性，即数据在内存中的地址应为 16 的倍数。在这部分代码设计中并未考虑数据对齐问题。数据存储不对齐，可能会影响 SIMD 指令的性能，导致性能下降。

2. 在代码中，多路向量并不能在每一次迭代中都能刚好处理完所有元素，则剩余元素需要通过串行的方式解决，这也会让代码的实际加速比并非与向量路数相同。

3. 代码中除能通过 simd 加速的代码外，还有一部分无法被加速的串行代码部分，这一部分为算法的加速设置了阈值，这同样使得算法实际加速比无法达到 2 或 4。

4. 访存问题可能也是影响性能的原因之一，串行算法与 simd 并行代码执行过程对于数据的访存是有所区别的，数据在缓存中的命中率不同也影响了并行算法的效率。

5. simd 串行算法虽然能够同时并行多条数据，但是每一条指令所消耗的时间比加长，这使得代码的指令时间开销变多，自然会影响最终的加速比。

进一步，我们将优化比数据可视化，可以看到如下的折线图，如图4.3所示：

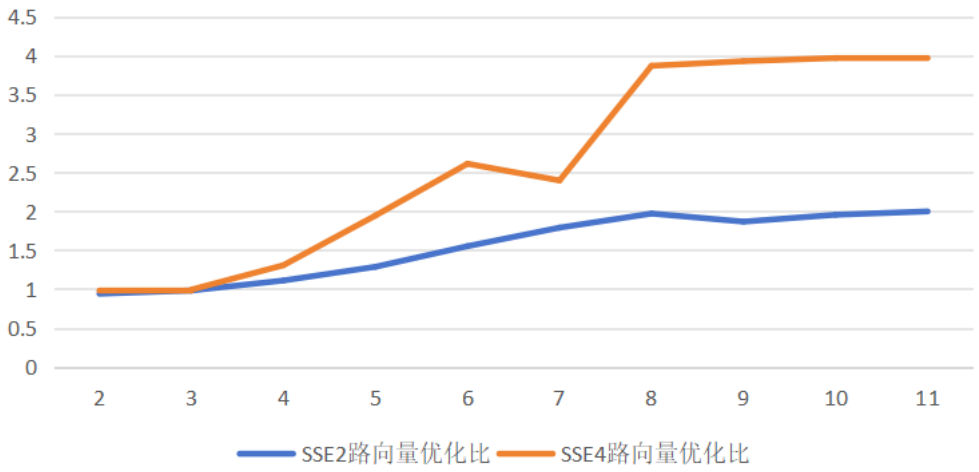

图 4.3: 优化比折线图

我们发现在矩阵规模比较小的时候，优化比也并不明显，甚至会慢于原来的串行代码，当矩阵规模逐渐变大，优化比也逐渐提高，并逐渐趋于稳定，根据 SIMD 并行化的特点，我分析得到以下可能的原因：这是因为串行算法虽然能够同时并行多条数据，但是每一条指令所消耗的时间比更长，。而指令消耗带来的额外开销甚至会比并行处理所能带来的优化时间更高，因而在矩阵规模比较小时，优化比也并不明显，甚至会慢于原来的串行代码；而当矩阵规模变大，这种额外的时间开销在并行优化的时间面前显得微不足道，因此优化比不断提高，当然，正如在上一环节提到的优化比阈值问题，矩阵规模不断变大，优化比会趋于稳定，并且低于所采用的向量路数。

4.1.2 串行算法与 AVX4 路优化、8 路优化的比较

在 x86 平台测试中，我通过对 LU 算法进行 AVX 优化，测试得到以下数据，如表4所示：

同样的，在 AVX 测试中也可以发现有着 simd 加速算法的实际优化比低于使用的向量路数（4 和 8）、并且在矩阵规模较小时优化效果不明显等问题。这些问题的原因与 SSE 优化在 x86 平台下表现的原因类同，这里就不再赘述。

4.1.3 SSE 与 AVX 方法的比较

得到以上在 x86 平台下对 LU 算法进行 SSE 和 AVX 并行优化的测试结果之后，我们可以根据实验数据对两种方法的效果进行更深入的对比分析。为优化目标的统一，我们选择 SSE 的 4 路向量优化

规模	x86 运行时间（ms）			规模	x86 运行时间（ms）		
	未优化	AVX			未优化	AVX	
		AVX4 路向量优化	AVX8 路向量优化			AVX4 路向量优化	AVX8 路向量优化
2	0.00734	0.0097151	0.00953006	7	10.2628	4.28415	3.45712
3	0.01438	0.0215518	0.0214538	8	73.5796	24.5066	16.5025
4	0.0437	0.0630095	0.0597216	9	572.377	161.835	97.8073
5	0.19859	0.210157	0.194872	10	4668.5	1182.69	698.543
6	1.27413	0.826983	0.700666	11	38710	9155.33	4736.08

表 4: 实验数据

与 AVX 的 4 路向量优化两种方法进行比较分析，以得到对 SSE、AVX 指令集的异同的分析。以下是 SSE 的 4 路向量优化与 AVX 的 4 路向量优化两种方法得到的优化比展示图，如图4.4所示：

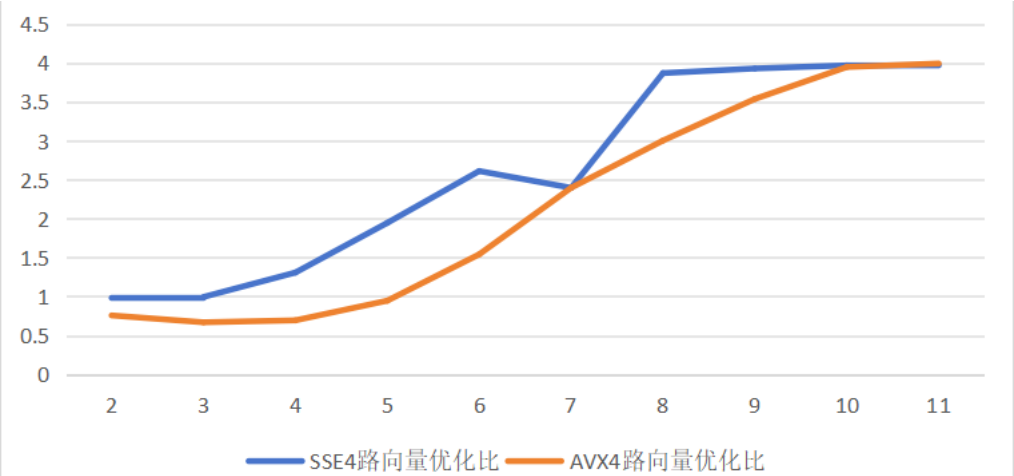

图 4.4: 优化比折线图

由上图我们可以发现 AVX 在矩阵规模比较小，优化比会比 SSE 更差，而随着矩阵规模增大，两种方法的优化比都逐渐提高，并且逐渐重合，趋于相同的稳定值。

通过对 SSE、AVX 指令集架构的研究，我分析了出现这种现象可能的原因：AVX 指令集相比于 SSE 指令集拥有更宽的数据寄存器（256 位），可以同时处理更多的数据。在处理大规模矩阵时，AVX 能够更充分地发挥其吞吐量高的优势，从而加快计算速度。

然而，在处理小规模矩阵时，由于数据量较小，AVX 指令并不能充分发挥其优势，甚至因为寄存器的宽度过大而导致填充造成的浪费，反而影响性能。与此相反，SSE 指令在处理小规模矩阵时，由于数据量更适合其寄存器宽度（128 位），表现较好。

因此，在性能测试中，随着矩阵规模增大，AVX 能够更好地展现其优势，而在小规模矩阵中，SSE 性能可能会优于 AVX。而最终，两者的性能会逐渐接近并趋于稳定的原因则又回到前面关于优化比阈值的分析。

4.1.4 SSE 部分优化与全部优化效果的比较

在 LU 算法中，主要有两部分可以进行并行加速，一部分是计算上三角矩阵（U），另一部分是计算下三角矩阵（L）。为进一步分析 SIMD 加速优化的特点，我通过对这两个部分分别加速，来对比这两个部分进行 SIMD 优化对程序速度的影响，同时将这两部分结果与全部优化的结果进行比较，希望得到一些有意思的发现。

测试数据如表5所示：

规模	x86 运行时间（ms）				规模	x86 运行时间（ms）			
	未优化	SSE				未优化	SSE		
		SSE_L（2）	SSE_U（2）	SSE2 路向量优化			SSE_L（2）	SSE_U（2）	SSE2 路向量优化
2	0.00734	0.007808	0.007512	0.00779	7	10.2628	8.0961	8.2799	5.7357
3	0.01438	0.01491	0.01503	0.01465	8	73.5796	56.0879	56.8138	37.3459
4	0.0437	0.04714	0.04286	0.03939	9	572.377	449.541	454.307	306.647
5	0.19859	0.19443	0.18954	0.15439	10	4668.5	3534.07	3637.26	2388.63
6	1.27413	1.08194	1.07977	0.82146	11	38710	30370.8	31065.8	19370.1

表 5: 实验数据

我们可以发现，对两个部分分别加速的结果类似，运行时间都很接近，说明这两个部分对程序速度的影响应该大致相同。

进一步，我们将分别加速的优化比与全部加速的优化比进行比较，如表6所示：

规模	2	3	4	5	6	7	8	9	10	11
分别加速优化比	0.9401	0.9645	1.0196	1.0477	1.17763	1.267	1.3046	1.2739	1.321	1.2746
全部加速优化比	0.94	0.9816	1.11	1.2863	1.5511	1.7893	1.9702	1.8666	1.9545	1.998

表 6: 分别加速的优化比与全部加速的优化比

我们发现，按照我们上面所说的 “两个部分对程序速度的影响应该大致相同”，那么每个部分对于全部优化的优化比的贡献占比应该为 1/2；也就是说，如果全部优化的优化比为 x，部分优化的优化比应该为 $\frac{2x}{x+1}$，但是我们发现实际上部分优化的优化比虽然能接近这个值，但是都会略低于这个值，这可能是因为：

在 LU 分解中，上三角矩阵和下三角矩阵的计算是相互依赖的过程，上三角矩阵的计算需要使用到下三角矩阵的值，而下三角矩阵的计算则需要使用到上三角矩阵的值。因此，如果单独使用 SIMD 方法并行优化上三角矩阵计算或下三角矩阵计算，由于无法实现完全并行化，仍然会存在数据依赖性，导致部分计算无法并行化，从而影响优化效果。

与此同时，如果同时并行化上三角矩阵和下三角矩阵的计算，可以更充分地利用并行计算的优势，减少串行部分的影响，从而获得更好的优化效果。因此，同时并行化的方式能够更有效地提高 LU 分解的计算效率，而单独并行优化上三角矩阵或下三角矩阵的方法可能无法达到全面的优化效果。

4.1.5 SSE 内存对齐与不对齐算法的效果比较

以下表7是使用 SSE 指令集设计的内存对齐与不对齐算法的优化比比较表：

规模	2	3	4	5	6	7	8	9	10	11
不对齐算法优化比/对齐算法优化比	1.0002	1.1014	1.0117	0.9923	1.0527	1.0119	0.9474	1.0087	1.0171	0.9812

表 7: 内存对齐与不对齐算法的优化比比较

我们可以看到这里对齐算法与不对齐算法的优化效果差不多，这可能跟测试矩阵的规模还不算很大有关，而通过网上查资料，为进一步了解到，这可能还与 CPU 和编译器有关：不同的 CPU 架构对内存对齐的需求和影响可能不同，有些 CPU 可能对内存对齐更为敏感，而有些 CPU 可能对内存访问的优化较为强大，从而减少了内存对齐的影响；编译器可能会自动对部分内存操作进行优化，包括一定程度上的内存对齐处理，因此对于一些简单的算法，编译器可能已经进行了一定程度的优化，减少了内存对齐所能带来的额外优势。

4.1.6 串行优化以及串行优化后的并行优化探索

在实现 LU 算法的过程中，我发现 LU 算法是基本将计算上三角矩阵和下三角矩阵的过程完全隔离，这可能会带来很大的时间开销，于是查询资料找到了优化的 LU 算法——Count_LU 算法。Count_LU 算法主要针对 LU 分解中的计算过程进行优化。相比传统的 LU 分解算法，Count_LU 算法在计算下三角矩阵和上三角矩阵时采用了一种更高效的计算方式，可以减少乘法和加法的次数，从而提高计算效率。

以下是 Count_LU 算法重点代码:

```
void croutLuDecomposition(vector<vector<double>>& mat, vector<vector<double>>& lower,
vector<vector<double>>& upper) {
    int n = mat.size();

    for (int i = 0; i < n; i++) {
        lower[i][i] = 1.0;

        for (int j = i; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < i; k++) {
                sum += lower[i][k] * upper[k][j];
            }
            upper[i][j] = mat[i][j] - sum;
        }

        for (int j = i + 1; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < i; k++) {
                sum += lower[j][k] * upper[k][i];
            }
            lower[j][i] = (mat[j][i] - sum) / upper[i][i];
        }
    }
}
```

Count_LU 算法通过优化计算下三角矩阵和上三角矩阵时的乘法和加法操作，减少了计算量，提高了计算效率。同时重新设计了计算流程，减少了数据之间的依赖关系，提高了并行度，从而加速计算过程。

根据实验测试数据，我们可以发现，在不同矩阵规模下，Count_LU 算法的优化效果可能会有所不同。对于较小规模的矩阵，由于数据量较少，优化效果可能相对较小，因为内存访问和计算开销相对较低。而对于较大规模的矩阵，Count_LU 算法的优化效果可能更为明显，因为传统的 LU 分解算法在大规模数据下容易出现计算瓶颈，而 Count_LU 算法的优化方法能够更有效地提高计算效率，加速计算过程。

除此之外，如果将 Count_LU 算法进行并行化处理，可以获得比直接并行化更好的效果。这说明

算法的优化需要综合考虑串行和并行两个方面，才能够实现最佳的效果。在串行方面，通过算法设计和代码优化来减少计算和内存访问的开销，提高效率；在并行方面，利用 SIMD 的并行来加速算法的执行。综合考虑这些方面，才能够使算法达到更高的性能和效率，实现最佳的优化效果。

4.2 ARM 平台下优化

在 ARM 平台上，我用在 x86 平台上类似的分析方法对 ARM 平台下的优化结果进行了全面分析，以下表8是基础串行代码与 NEON 加速代码在 ARM 平台下运行测试结果：

规模	ARM 运行时间（ms）		规模	ARM 运行时间（ms）	
	未优化	NEON		未优化	NEON
2	0.00097413	0.00101393	7	1.01232	0.354853
3	0.0016665	0.00173628	8	8.46296	2.76314
4	0.00420614	0.00388603	9	69.7063	19.7401
5	0.0190396	0.0130965	10	619.605	172.932
6	0.12656	0.0584534	11	5763.13	1664.51

表 8: ARM 平台下测试数据

根据数据结果，我们发现在 ARM 平台下，NEON 的优化结果也存在着随矩阵规模变大逐渐趋于稳定、且低于所使用的 4 路向量的路数。这个现象的原因与 x86 平台下遇到的相同问题原因基本相同。

不仅如此，根据数据分析发现，在不同指令集架构下，对于部分优化与全部优化的区别问题、内存对齐问题、以及使用不同路向量优化效果不同的问题大同小异，造成这些现象的原因也如上 x86 平台下的分析结果相同，这里就不再累赘地在 ARM 平台上对这些问题进行讨论。

4.3 综合结果分析

此部分将通过 x86 平台与 ARM 平台下相同加速比的 SSE/AVX 与 NEON 的优化进行对比，探讨不同架构下的指令集的类同与区别。

为保证分析的合理性，需要保证我们选择的多路向量的路数是相同的，所以我们在此选择 SSE 的 4 路向量优化与 NEON 的 4 路向量优化进行比较，以分析不同平台指令集的关系。

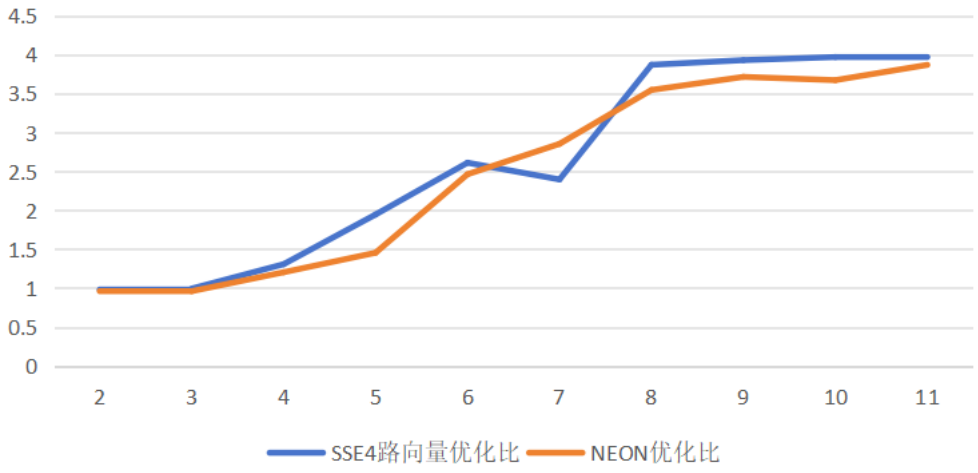

图 4.5: 优化比折线图

根据上图，我们发现两种方式的优化效果非常相似，这可能与这两者本身极其相似的设计和用途有关；

1. 首先，SSE 和 NEON 都是针对向量化计算而设计的指令集架构;
2. SSE 指令集包括多达 128 位的 SIMD 指令，NEON 指令集支持不同长度的 SIMD 指令，包括 64 位和 128 位。而在这次的问题解决中，我使用了 128 位的 NEON 指令。位数相同也可能是导致两种优化效果相似的原因之一。

5 小组实验总体分析

当选择线性方程组求解方法时，高斯消去算法和 LU 分解算法各有优势：
高斯消去算法：

- 用途：解决一般的线性方程组。
- 优势：简单直接，适用于规模较小的问题。
- 限制：可能存在数值不稳定性，特别是在主元接近零时。

LU 分解算法：

- 用途：解决多个具有相同系数矩阵但不同右端项的方程组。
- 优势：提高了计算效率，有利于数值稳定性的分析。
- 限制：可能遇到奇异性问题，需要额外的处理。

6 个人任务完成总结

本次实验中，我们小组完成了对于 Guass 和 LU 算法的 simd 加速，并对两种算法的特点与用途进行分析。

在我的任务部分，我完成了对 LU 算法的 SIMD 并行优化实验，在 x86 和 ARM 平台上进行了实验，并对算法性能进行了全面分析。实验中，我成功实现了对矩阵的每一行进行向量乘法和加法操作，并通过 SIMD 指令并行计算矩阵的每一行，加快了 LU 分解的速度，提高了算法的计算效率。

在实验中，我通过对比串行和并行算法的运行时间和性能表现，深入分析了 SIMD 并行优化对 LU 算法性能的影响。探索发现并行算法的性能相关性质，以及充分利用串行与并行方法，提高算法的运行速度。同时也注意到在并行算法中需要合理划分和处理数据，并且要充分利用 SIMD 指令进行向量化操作，以实现并行加速的效果。

通过本次实验，我进一步加深了对并行计算和优化技术的理解，提高了算法的性能和效率。

总的来说，本次实验任务完成度较高，成功实现了对 LU 算法的 SIMD 并行优化，并且对算法性能进行了全面分析。通过实验，我不仅提高了对并行计算和优化技术的认识，也为进一步的算法优化和并行计算研究打下了基础。