南開大學
Nankai University

计算机学院

SIMD 编程实验报告

Grobner 基计算中的高斯消去

姓名：卢艺晗

学号：2213583

专业：计算机科学与技术

2024 年 4 月 28 日

目录

1 综述

1.1 摘要

本次 SIMD 编程实验，围绕高斯消元法选题，主要进行了以下探究：在 ARM 和 x86 平台上，使用 NEON、SSE、AVX 三种指令集分别进行 SIMD 向量化，经过测试结果分析，寻找到对该算法性能优化效果最好的是在 x86 平台上使用 AVX 进行向量化。并对 AVX 进一步探究了不同算法策略对性能的影响，包括对齐与不对齐的对比，对二重循环部分向量化、对三重循环部分向量化和全部进行向量化的对比，并根据测试结果，分析影响性能的因素，为期末选题的研究奠定基础。本次实验，还使用 VTune 深入剖析了 x86 平台上各种算法实现的性能。最后，基于在普通高斯消元法 SIMD 向量化得到的经验，对 Grobner 基计算中的高斯消去进行了 x86 平台上串行算法的实现和 SIMD 向量化加速的实验，并取得一定成果。

1.2 实验平台信息

指标	值
CPU 型号	Intel® Core™ i7-14650HX
CPU 核数	16
CPU 主频	2.20GHz
L1/L2/L3 大小	1.4MB/24.0MB/30.0MB
内存容量	32.0GB
操作系统及版本	windows11 家庭中文版
编译器及版本	Code::Blocks 20.03 gcc 8.1.0
AVX 编译选项	-mavx

表 1: x86 实验平台信息

指标	值
服务器	鲲鹏服务器
编译器	BiSheng 编译器
编译器版本	2.1.0
操作系统	linux

表 2: ARM 实验平台信息

1.3 选题简介

普通高斯消元：高斯消元法（也称高斯消去法）是求解线性方程组的一个基本方法, 它可以用于决定线性方程组的解、矩阵的秩, 以及可逆矩阵的逆。高斯消元法是通过矩阵的行变换达到消元的目的, 从而将方程组的系数矩阵转化为三角矩阵, 最后获得方程组的解, 其主要包括 3 个过程: 输入数据, 消元和回代求解。

在本实验中，对普通高斯消元算法的并行化研究主要针对 SIMD 向量化。分别在 ARM 和 x86 平台，采用 NEON、SSE、AVX 三种不同指令集，探寻对该算法性能优化效果最好的指令集和算法策略，并深入剖析性能的印象因素，为期末研究奠定初步基础。普通高斯消元法的串行算法思路可用伪代码表示如下。本实验的 SIMD 并行优化算法均在此基础上进行向量化。

Algorithm 1 普通高斯消元法的串行算法

```
1:  n:=size(A)
2:  //消去过程
3:  for k := 1  to  n do
4:      for i := k+1  to  n do
5:          factor := A[i,k] / A[k,k]
6:          for j :=k+1 to n do
7:              A[i,j] :=A[i,j]-factor*A[k,j]
8:          end for
9:          b[i] :=b[i]-factor*b[k]
10:     end for
11: end for
12: //回代过程
13: x[n] :=b[n]/A[n,n]
14: for i :=n-1 to 1  do
15:     sum :=b[i]
16:     for j :=i+1 to n do
17:         sum:=sum-A[i,j]*x[j]
18:     end for
19:     x[i] :=sum/A[i,i]
20: end for
```

特殊高斯消去：本实验选取 Grobner 基计算中的高斯消去这一特殊的高斯消元过程，进行串行算法的实现和 x86 平台上的 SIMD 并行化实现。与普通高斯消取相比，该特殊高斯消去有三点不同之处：

（1）矩阵元素的值只能是 0 或 1；

（2）Grobner 基计算中的高斯消去只有异或运算：加法运算实际为异或运算（0+0=0，0+1=1，1+0=1，1+1=0）；减法运算为加法的逆运算，亦为异或运算；乘法运算（0*0=0，0*1=0，1*0=0，1*1=0）实际可消。

（3）矩阵行分为两类：“消元子”和“被消元行”，在输入时给定：在消去过程中，“消元子”充当减数，所有消元子首项位置均不同，但不涵盖所有对角线元素；“被消元子”充当被减数，若恰好包含消元子中缺失的对角线 1 元素，则升格为消元子。

1.4 github 链接

本实验的代码已经上传到 GitHub 的仓库上。**源码 GitHub 链接：** GitHub/Lucyannn/Parallel Design

2 x86、ARM 平台上高斯消元法 SIMD 并行化实现

2.1 算法设计

分别在 x86 平台上采用 SSE、AVX 语言，在 ARM 平台上采用 NEON 语言，实现普通高斯消去的 SIMD 并行化。基本实现采用 Intrinsic 函数进行代码编写，对高斯消去过程中的二重循环部分、三重

循环部分均进行向量化，并分别与 x86、ARM 平台上使用串行算法执行的时间做对比，以探究 SIMD 并行加速的效果。

普通高斯消去核心逻辑： 伪代码如下：SSE 和 NEON 均采用 4 路向量化，伪代码如下，而 AVX 采用 8 路向量化，实现的伪代码类似。

测试用例生成： 本实验设计了多个不同规模的矩阵作为测试样例。为了保证鲁棒性和避免计算结果出现无穷，采用随机数来辅助测试样例生成，且直接生成上三角矩阵。测试样例生成的代码如下。共生成了 100、250、500、750、1000、2000、3000、4000 这 8 中规模的矩阵进行测试。

Algorithm 2 SIMD Intrinsic 版本的普通高斯消元

Data: 系数矩阵 A[n,n]
Result: 上三角矩阵 A[n,n]
for k = 0 **to** n-1 **do**
    $vt \leftarrow$ dupTo4Float($A[k,k]$)
    for $j = k+1; j+4 \le n; j+=4$ **do**
        $va \leftarrow$ load4FloatFrom(&$A[k,j]$)
        $va \leftarrow va/vt$
        store4FloatTo(&A[k,j], va)
    end for
    for all j **in** 剩余所有下标 **do**
        $A[k,j] \leftarrow A[k,j]/A[k,k]$
    end for
    $A[k,k] \leftarrow 1.0$
    for i = k+1 **to** n-1 **do**
        $vaik \leftarrow$ dupToVector4($A[i,k]$)
        for $j = k+1; j+4 \le n; j+=4$ **do**
            $vakj \leftarrow$ load4FloatFrom(&$A[k,j]$)
            $vaij \leftarrow$ load4FloatFrom(&$A[i,j]$)
            $vx \leftarrow vakj * vaik$
            $vaij \leftarrow vaij - vx$
            store4FloatTo(&A[i,j], vaij)
        end for
        for all j **in** 剩余所有下标 **do**
            $A[i,j] \leftarrow A[i,j] - A[k,j] * A[i,k]$
        end for
        $A[i,k] \leftarrow 0$
    end for
end for

Listing 1: 普通高斯消去测试用例生成

```
float m[N][N];
void m_reset() {
    for(int i=0; i<N; i++) {
        m[i][j] = 0;
        m[i][i] = 1.0;
        for(int j=i+1; j<N; j++)
            m[i][j] = rand()%1000;
    }
    for(int k=0; k<N; k++)
        for(int i=k+1; i<N; i++)
            for(int j=0; j<N; j++)
```

```
                m[i][j] += m[k][j];
                m[i][j]=(int)[i][j]%1000;
}
```

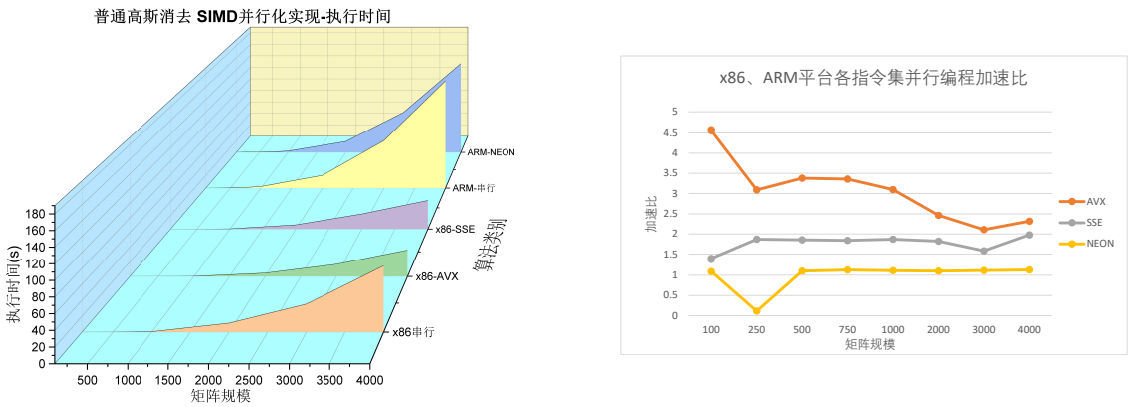
2.2 测试结果

在 x86 平台上的 windows 环境中，采用 QueryPerformance ()，在 ARM 平台上的 linux 环境中，采用 getimeofday()，精确测量程序执行时间，如表 3 所示。

	100	250	500	750	1000	2000	3000	4000
x86 平台								
串行算法	**0.0012187**	**0.009556**	**0.078514**	**0.257528**	**0.612596**	**11.2924**	**35.1688**	**83.2515**
AVX	**0.0002675**	**0.003093**	**0.02322**	**0.076677**	**0.197864**	**4.58894**	**16.6919**	**35.9754**
SSE	**0.0008744**	**0.005117**	**0.042353**	**0.140095**	**0.328075**	**6.20029**	**22.2076**	**42.089**
ARM 平台								
串行算法	**0.0024897**	**0.038826**	**0.309999**	**1.081826**	**2.543269**	**20.2207**	**75.02162**	**169.4372**
NEON	**0.002279**	**0.35379**	**0.281817**	**0.959877**	**2.28479**	**18.363**	**67.2514**	**149.909**

表 3: x86、ARM 平台高斯消去 SIMD 并行化执行时间（单位:s）

执行时间的直观展示如图 1(a) 所示。根据表格数据，计算各种指令集对普通高斯消去的 SIMD 向量化加速比，如图 1(b) 所示。



(a) x86、ARM 平台实现普通高斯消去的 SIMD 并行化执行时间（单位：s）

(b) x86、ARM 平台实现普通高斯消去的 SIMD 并行化加速比

图 2.1: x86、ARM 平台实现普通高斯消去的 SIMD 并行化

2.3 结果分析

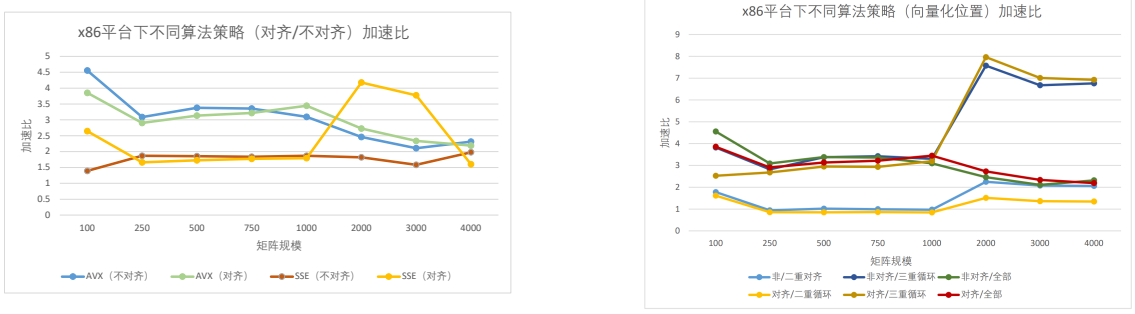
从表3和图1(a)可以看出，AVX、SSE、NEON 三种指令集并行化均比串行算法执行的性能更好，其中 x86 平台的整体执行效率高于 ARM 平台。在 x86 平台上，分别用 AVX 和 SSE 执行，AVX 的性能优化程度高于 SSE。具体分析如下：

x86 平台上，SSE 提供 128 位宽的寄存器，最多同时处理 4 个单精度浮点数；而 AVX 拓展到了 258 位宽的寄存器，最多同时处理 8 个单精度浮点数。通过向量化并行加速，AVX 的加速效果更高。

在矩阵规模较小（<150）时，向量化指令在全部中占比相对较高，并行优势体现明显，AVX 的加速比超过了 4。随着矩阵规模增大，AVX 加速比下降，与 SSE 的加速比越来越接近，原因可能是需要执行的指令数量增加，总体耗时具有一定开销。但经过 SIMD 并行化的程序性能相对串行算法仍然有明显提升。通过 VTune 分析，SIMD 并行化后，Effective Physical Core Utilization 均有所增加。

ARM 平台上，NEON 算法的加速比相对 AVX、SSE 并不算高，原因可能是 NEON 指令集本身常用作低能耗优化方法，自身开销对性能有一定的限制，但其也实现了程序性能的提升。随着矩阵规模增加，并行部分占比增加，导致加速比呈缓慢上升趋势。

3 不同算法策略对比



(a) x86 平台不同算法策略（对齐/不对齐）加速比

(b) x86 平台不同算法策略（向量化位置）加速比

图 3.2: x86 平台不同算法策略的加速比

3.1 对齐/不对齐

从内存中读取数据时，数据在内存中是否对齐，对于读取数据的速度也有影响。现代处理器通常以字（通常是 4 或 8 字节）为单位进行数据访问。当数据按照其自然对齐界限（如整数在 4 字节地址上对齐，双字节类型在 8 字节地址上对齐等）存储时，处理器可以在一个操作周期内有效地加载和存储这些数据。如果数据未对齐，处理器可能需要执行额外的内存访问：首先读取第一个部分覆盖的内存块，然后读取下一个内存块，最后组合这两部分的数据以获取完整的值。这增加了访问延时和额外的处理负担。因此，内存中数据对齐和不对齐，对于程序的执行效率也有影响。

在 x86 平台上，分别使用 SSE 和 AVX 进行对齐、不对齐策略的对比，其中 SSE 进行 16 字节对齐，AVX 进行 8 字节对齐。记录程序执行时间如表4 所示，并分别计算加速比如图 2(a)所示。

实现细节:在对齐策略下,需要使用对齐指令取代非对齐指令(如用 _mm_load_ps 取代 _mm_loadu_ps 等)，以及通过 _mm_maloc() 函数为矩阵数据分配对齐的内存。在执行高斯消去算法读取数据并运算的时候，需要先串行运算到对齐位置，再并行对齐读取，最后对该行剩下的元素进行串行运算，才能保证计算完整、正确。关键部分的代码如下（以 AVX 二重循环部分的向量化为例）:

Listing 2: 对齐处理关键流程代码示例

```
for (int k = 0; k < N; k++) {
        __m256 vt = _mm256_set1_ps(A[k][k]);
        int j=k+1;
```

```
        //先串行处理到对齐位置
         while(reinterpret_cast<uintptr_t>(&A[k][j])%32){
            A[k][j]/=A[k][k];
            j++;
        }
        //向量化 对齐
        for (j; j + 8 <= N; j += 8) {
            __m256 va = _mm256_load_ps(&A[k][j]);
            va = _mm256_div_ps(va, vt);
            _mm256_store_ps(&A[k][j], va);
        }
        //串行处理剩余的元素
        for (; j < N; j++) {
            A[k][j] /= A[k][k];
        }
        A[k][k] = 1.0f;
```

矩阵规模	100	250	500	750	1000	2000	3000	4000
串行算法	0.0012187	0.0095557	0.078514	0.257528	0.612596	11.2924	35.1688	83.2515
AVX（不对齐）	0.0002675	0.0030927	0.02322	0.076677	0.197864	4.58894	16.6919	35.9754
AVX（对齐）	0.0003163	0.0032901	0.025034	0.080096	0.177789	4.14133	15.063	38.0166
SSE（不对齐）	0.0008744	0.0051174	0.042353	0.140095	0.328075	6.20029	22.2076	42.089
SSE（对齐）	0.000461	0.0057439	0.045447	0.14508	0.341549	2.70329	9.31458	51.9594

表 4: x86 平台不同算法策略（对齐/不对齐）执行时间（单位：s）

3.2 不同位置向量化

普通高斯消元算法中，影响程序效率的重点之一为循环部分。考虑到高斯消去部分的二重循环、三重循环部分均可进行向量化，为了进一步分析影响程序性能的因素，分别单独对这两部分进行向量化处理，进行对比。在 x86 平台上使用 AVX 语言，分别在对齐策略、不对齐策略下进行测试，记录执行时间如表 4 所示，并计算加速比如图2(b)所示。

矩阵规模	100	250	500	750	1000	2000	3000	4000
串行算法	0.0012187	0.0095557	0.078514	0.257528	0.612596	11.2924	35.1688	83.2515
非对齐								
二重循环	0.0006858	0.0101638	0.077531	0.259111	0.631077	5.02271	16.9567	40.4673
三重循环	0.0003192	0.0033903	0.023248	0.075316	0.18565	1.49093	5.26672	12.3118
全部	0.0002675	0.0030927	0.02322	0.076677	0.197864	4.58894	16.6919	35.9754
对齐								
二重循环	0.000754	0.0111883	0.092511	0.299123	0.725926	7.46219	25.856	61.9502
三重循环	0.0004829	0.00357	0.026594	0.08777	0.191893	1.41841	5.01731	12.0226
全部	0.0003163	0.0032901	0.025034	0.080096	0.177789	4.14133	15.063	38.0166

表 5: x86 平台不同算法策略（向量化位置）执行时间（单位：s）

3.3 结果分析

对齐/不对齐策略对比：

对于 AVX，中小规模的矩阵（<=750）在对齐策略下，性能并未高于不对齐策略，可能原因是采用 256 位寄存器的 AVX 由于数据随机生成，在未到对齐位置执行串行计算的几率更大，造成了相对不对齐的全部并行操作，耗时更多的结果。在大规模矩阵 (750 4000) 下，数据操作量增加，由于对齐操作减少不必要的数据操作，因此作用愈加明显，表现出了对齐的优势。

对于 SSE，矩阵规模在 100 左右时，对齐策略由于减少不必要的数据操作，加之规模小，本身数据操作总数较少，对齐策略对数据操作数量减少得效果更明显，因此加速比达到了接近 2，远高于不对齐策略下的 SSE。矩阵规模 <=1000 左右时，对齐策略加速比不如非对齐策略，可能原因类似 AVX；在矩阵在 1000 4000 的大规模下，对齐策略加速比出现了显著提升，部分原因可能由于实验的偶然性，但主要原因与 AVX 的分析类似。参见图2(a)。

因此，选择对齐/不对齐策略对于性能提升的影响，在本实验中，与矩阵的规模有关。考虑到矩阵生成的算法，也可能与测试数据的总体规律有一定联系。这一点将会在未来进一步探究。

不同位置向量化对比：

在不同位置向量化的实验中，对齐和不对齐策略下均表现出了相近的特征：由于三重循环部分在执行时的操作次数更多，对程序执行时间的影响显著高于二重循环部分的影响，因此仅对三重循环向量化的策略性能始终远高于仅对二重循环向量化的策略性能。

在矩阵规模较小时（<1000），性能测试结果为仅对三重循环向量化和全部向量化的性能加速比相近，均高于仅对二重循环向量化。原因是矩阵规模较小时，向量化操作数量总体占比较大，而三重循环部分在其中的占比又比二重循环大，对性能影响显著，因此全部向量化时和三重循环向量化的效果相近。在矩阵规模较大时（>1000），性能测试结果为仅对二重循环向量化和全部向量化的性能加速比相近，均远低于于仅对二重循环向量化。可能原因仍有待探究。

4 VTune 测试程序性能

在 x86 平台上，使用 VTune 深入测试并剖析程序性能。重点测试 Gauss Elimination（即高斯消去部分）的 CPU time、CPI、内存操作和 P-core Retiring 等数据。详细数据记录如表6。

根据 VTune 测试数据对各个问题的分析已经在前面分别提到，在此不再赘述。

	GaussElimination					Total
	CPUtime	**Clockticks**	**Instructions Retired**	**CPI**	**Pcore Retiring**	**MemoryOperations**
串行	**69.708**	**2.68814E+11**	**1.34179E+12**	**0.2**	**85.50%**	**31.40%**
AVX_1	**13.291**	**52589020000**	**1.8329E+11**	**0.287**	**60.30%**	**29.20%**
SSE_1	**16.842**	**75407487000**	**2.71893E+11**	**0.277**	**58.50%**	**30.20%**
AVX_2	**2.739**	**2559302000**	**3897009000**	**0.657**	**61.20%**	**29.10%**
SSE_2	**19.222**	**74872888000**	**2.78183E+11**	**0.269**	**56.90%**	**30.10%**
AVX_1_1	**68.814**	**2.66342E+11**	**1.34149E+12**	**0.199**	**83.70%**	**31.40%**
AVX_1_2	**13.729**	**52811608000**	**1.82898E+11**	**0.289**	**57.80%**	**29.10%**
AVX_2_1	**67.37**	**2.66201E+11**	**1.3413E+12**	**0.198**	**84.80%**	**31.30%**
AVX_2_2	**12.953**	**50261982000**	**1.81708E+11**	**0.277**	**61.90%**	**29.20%**

表 6: x86 平台 VTune 剖析关键数据

5 特殊高斯消去的 SIMD 并行化实现

5.1 算法设计

对 Grobner 基计算中这种特殊的高斯消去，分别设计串行算法和 SIMD 向量化的并行算法，在 x86 平台上进行性能的比较。采用 11 个不同规模的测试样例进行测试。测试样例信息如表 8 所示。

Grobner 数据集	矩阵列数	非零消元子行数	被消元行行数
例 1	130	22	8
例 2	254	106	53
例 3	562	170	53
例 4	1011	539	263
例 5	2362	1226	453
例 6	3799	2759	1953
例 7	8399	6375	4535
例 8	23045	18748	14325
例 9	37960	29304	14921
例 10	43577	39477	54274
例 11	85401	5724	756

表 7: Grobner 特殊高斯消去测试样例信息表

5.1.1 串行算法

设计特殊高斯的串行算法，主要流程包括数据读取、矩阵稀疏存储和稠密存储形式的转换、高斯消去、计算结果和记录程序执行时间。其中高斯消去部分主要逻辑为：分批次读取被消元行，逐一判断消元子和被消元行能否消元，若被消元行变为空行，则使用 clear() 清空；若被消元行在该批次内无对应消元子，则升格为消元子；若首项不被当前批次覆盖，则该批次结束。执行所有批次后，得出结果转换为稀疏矩阵。核心逻辑如下伪代码所示：

Algorithm 3 Grobner 基计算中的高斯消去核心逻辑

```
1: procedure GaussianEliminationBatch(eliminators, eliminated)
2:     newEliminators ← {} // 用于存储新生成的消元子
3:     for each row in eliminated do
4:         leadingOneIndex ← −1 //初始化首个非零元素的索引为 -1
5:         UpdateLeadingOne(row, leadingOneIndex) //初始化首个非零元素的索引为 -1
6:         while leadingOneIndex ≠ −1 do //循环直到当前行全为 0
7:             if leadingOneIndex is in iToBasis then
8:                 eliminator ← iToBasis[leadingOneIndex] //如果 leadingOneIndex 对应的消元
   子存在
9:                 for each i in row.size() do
10:                    oldValue ← row[i]
11:                    row[i] ← row[i]⊕ eliminator[i] //执行异或操作
12:                if oldValue ≠ row[i] then //如果元素发生变化
13:                    isChanged ← true //退出当前循环
14:                end if
15:
16:                if !isChanged then
17:                    break
18:                end if
19:                UpdateLeadingOne(row, leadingOneIndex)
20:            else//如果 leadingOneIndex 对应的消元子不存在
21:                iToBasis[leadingOneIndex] ← &row //将当前行升格为消元子
22:                newEliminators.push_back(row)
23:                break
24:            end if
25:        end while
26:        if leadingOneIndex == -1 then //如果当前行全为 0
27:            row.clear() //清空当前行
28:        end if
29:    end for
30:    for each newElim in newEliminators do
31:        eliminators.push_back(newElim)
32:
33: end procedure
34: procedure UpdateLeadingOne(row, leadingOneIndex)
35:    leadingOneIndex ← −1
36:    for i = 0 to row.size() - 1 do
37:        if row[i] == 1 then
38:            leadingOneIndex ← i
39:            break
40:        end if
41:    end for
42: end procedure
```

5.1.2 SIMD 向量化

对特殊高斯消去进行 SIMD 编程，主要在加载和存储数据、执行异或操作以及检查位变化操作中，进行了向量化。算法设计的伪代码如下：

Algorithm 4 Grobner 基计算中的高斯消去 SIMD 算法

```
1: procedure GaussianEliminationBatchAVX(eliminators, eliminated)
2:     newEliminators ← []
3:     for all rowineliminated do
4:         leadingOneIndex ← −1
5:         UpdateLeadingOne(row, leadingOneIndex) //更新首个非零元素的位置
6:         while leadingOneIndex ≠ −1 do
7:             it ← find(iToBasis, leadingOneIndex)
8:             if it ≠ end(iToBasis) then
9:                 row_vec ← _mm256_setzero_si256 //初始化 row 向量
10:                elim_vec ← _mm256_loadu_si256(it → second → data()) //加载消元子向量
11:                isChanged ← false
12:                for i ← 0torow.size()step8 do
13:                    row_vec ← _mm256_loadu_si256(&row[i]) //加载 row 向量
14:                    // 执行异或操作
15:                    row_vec ← _mm256_xor_si256(row_vec, elim_vec)
16:                    // 存储修改后的 row 向量
17:                    _mm256_storeu_si256(&row[i], row_vec)
18:                    // 检查是否有任何位发生变化
19:                    diff ← _mm256_xor_si256(_mm256_loadu_si256(&row[i]), _mm256_loadu_si256(&oldRo
20:                    if _mm256_movemask_epi8(diff) ≠ 0 then
21:                        isChanged ← true
22:                    end if
23:                end for
24:                if not isChanged then
25:                    // 如果没有任何改变，避免无限循环
26:                    break
27:                end if
28:                UpdateLeadingOne(row, leadingOneIndex) //再次更新首个非零元素的位置
29:            else
30:                // 将当前行升级为新的消元子
31:                iToBasis[leadingOneIndex] ← address_of(row)
32:                newEliminators.append(row)
33:                exit loop //退出循环
34:            end if
35:        end while
36:        // 检查行是否全为 0
37:        if leadingOneIndex = -1 then
38:            row.clear() //清空行
39:        end if
40:    end for
41:    // 将新消元子加入到现有消元子列表中
42:    for all newElim in newEliminators do
43:        eliminators.append(newElim)
44:    end for
45: end procedure
```

5.2 测试结果

在 x86 平台上，分别使用 SSE 和 AVX 对 Grobner 基计算中的高斯消去进行编程实现，记录程序执行时间，如表8所示。并计算了加速比，如图5.3所示。

	串行算法	AVX	SSE		串行算法	AVX	SSE
例 1	0	0	0	例 7	124.3467	85.26131	101.196
例 2	0.015075	0.005025	0.01005	例 8	935.1809	619.2864	770.0905
例 3	0.015075	0.005025	0.01005	例 9	1406.839	933.7638	1120.111
例 4	0.407035	0.231156	0.311558	例 10	4412.322	2905.643	3530.367
例 5	1.78392	1.19598	1.467337	例 11	2.437186	0.994975	1.763819
例 6	20.09548	13.58794	16.89447				

表 8: Grobner 特殊高斯消去执行时间 (单位：s)

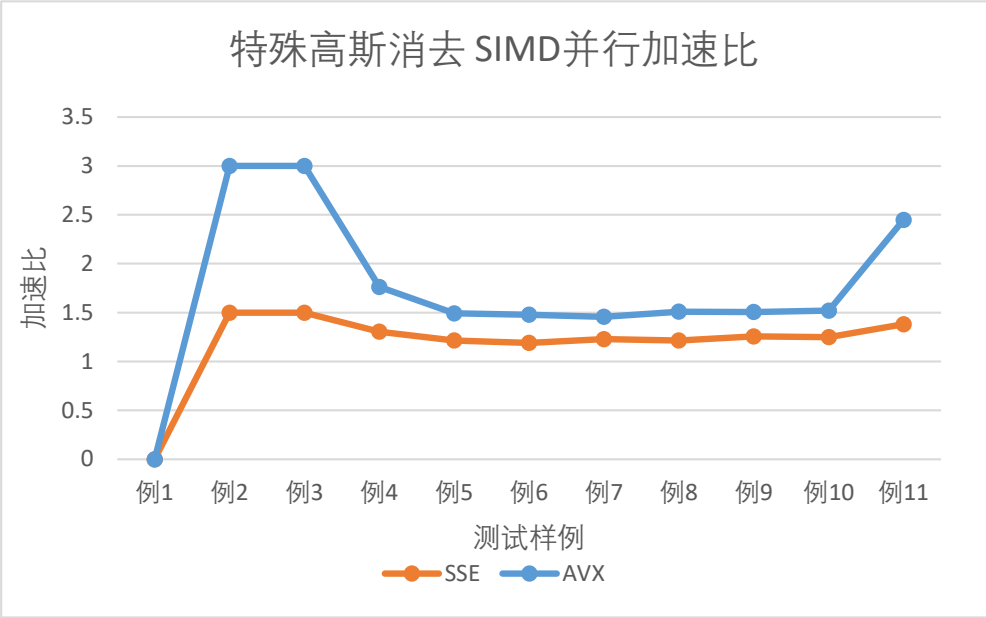


图 5.3: Grobner 特殊高斯消去 SIMD 加速比

5.3 结果分析

x86 平台上，AVX 和 SSE 均实现了对 Grobner 基计算中的高斯消去的并行加速。其中 AVX 的加速比始终高于 SSE。结合表 7 的测试用例规模信息，和图5.3 中的加速比图像可知，在消元子、被消元行数量较小时，AVX 加速比远高于 SSE，原因是需要操作的数据更少，此时向量化部分占比更大，同时处理 8 个单精度浮点数的 AVX 比同时处理 4 个单精度浮点数的 SSE 表现出更大优势。而当规模较大时，需要操作的数据量本身较大，且 Intrinsic 函数自身具有一定开销，AVX 的加速效果收到限制，和 SSE 的效果更加相近。

注：图 5.3 中，”例 1 “处的数据加速比均为 0，是由于该测试样例下执行时间极短，测得时间均趋近于零，故在计算加速比时，为该处数据赋值为 0.

6 总结

6.1 该子问题与期末研究的联系

本次实验为期末研究问题的 SIMD 并行化方向奠定了初步基础。通过普通高斯消元法在不同平台、不同指令集、不同算法策略下的对比分析，探究实现性能优化的更好策略。同时，初步实现了 Grobner 基计算中高斯消去的 SIMD 并行化的实现，性能得到了一定程度的优化。不过对于该特殊高斯消去的探究还不够深入，在期末研究中，将实现对特殊高斯消去进行向量化部分的策略选择，和对该问题本身逻辑实现的优化。

6.2 感悟：收获和不足

通过本次实验，我收获了很多。包括但不止对于高斯消去性能优化探究的理解、对 Origin 绘图、鲲鹏服务器命令操作的熟练掌握、对使用 VTune 剖析程序性能的进一步理解。

不过在分析测试结果，探索深入原因时，我感到自己在分析问题方面欠缺经验。对于实验结果产生的原因的深入、准确的剖析，离不开更底层知识的支撑和理解。因此，未来我还需进一步了解计算机的组成结构、并行加速的相关知识，来更好地分析问题、探索问题的更优解。