南開大學
Nankai University

计算机学院

SIMD 编程

SIMD 编程实现两种高斯消元算法

姓名：郭军凯

学号：2213627

专业：计算机科学与技术

2024 年 4 月 27 日

目录

1 问题描述 **2**
1.1 高斯消元 2
1.1.1 消元法 2
1.1.2 高斯消元 2
1.2 Grobner 基计算中的高斯消元 4
1.2.1 Grobner 基理论介绍 4
1.2.2 Grobner 基生成算法 5
1.2.3 Grobner 基生成算法中的高斯消元 6

2 普通高斯消元部分 **7**
2.1 SIMD 算法设计与复杂度分析 7
2.2 结果展示与分析 8
2.2.1 x86 架构（SSE、AVX、AVX512 编程） 9
2.2.2 arm 架构（NEON 编程） 10

3 Grobner 基计算中的高斯消元部分 **10**
3.1 SIMD 算法设计与复杂度分析 10
3.2 结果展示与分析 11
3.2.1 x86 架构（SSE、AVX、AVX512 编程） 11
3.2.2 arm 架构（NEON 编程） 12

4 实验环境与仓库链接 **13**
4.1 实验环境 13
4.2 仓库链接 13

1 问题描述

1.1 高斯消元

1.1.1 消元法

消元法是将方程组中的一方程的未知数用含有另一未知数的代数式表示，并将其带入到另一方程中，这就消去了一未知数，得到一解；或将方程组中的一方程倍乘某个常数加到另外一方程中去，也可达到消去一未知数的目的。

$$\begin{cases} 4x + y = 100 \\ x - y = 100 \end{cases}$$

如上例所示，将方程组中两方程相加，消元 y 可得：$5x = 200$, 即 $x = 40$

消元法理论的核心主要如下:

- 两方程互换，解不变；
- 一方程乘以非零数 k，解不变；
- 一方程乘以数 k 加上另一方程，解不变。

1.1.2 高斯消元

高斯在这些结论的基础上，提出了高斯消元法，首先将方程的增广矩阵利用行初等变换化为行最简形，然后以线性无关为准则对自由未知量赋值，最后列出表达方程组通解。高斯消元法（Gauss–Jordan elimination）是求解线性方程组的经典算法，它在当代数学中有着重要的地位和价值，是代数学的重要基础理论之一。高斯消元的主要过程如下。

- 线性方程组矩阵化

  写出线性方程组的增广矩阵，待后续过程使用。

- 消去

  - 寻找主元，找到首个存在非零元素的列，并将该元素所在的行交换到目前正在处理的行上。
  - 利用矩阵初等变换，将主元变化为 1（该行同时乘以主元的逆元）。
  - 利用矩阵初等变换，将后面的行该列的元素变为 0（减去若干倍目前正处理的行）。

  经过消去后，可以得到一个主元均为 1 的行阶梯型矩阵。

- 回代

  类似消去的过程，再次利用矩阵变换，消去主元所在列其他位置的数字，得到一个行最简型矩阵。

- 表示方程组解系

  根据消元和回代后得到的行最简型矩阵，写出对应其次线性方程组的基础解析，和原方程组的一个特解，根据这些解向量表示原方程组的解系。

根据高斯的研究结论，计算机学家提出了一种使用计算机解决线性方程组的算法，直接模拟上述过程计算线性方程组的解。

可以发现，直接模拟上述流程，消元和回代过程的时间复杂度为 $O(n^3)$，但是，工程实践中，往往不需要求出方程组的整个解系，只需要求出任何一个特殊解或者满足某种条件的特殊解，回代过程可以使用固定数值回代代替，可以优化为 $O(n^2)$，因此，后续的讨论将围绕着消元部分进行，不再讨论回代部分。消元部分的伪代码如下。

Algorithm 1 Gaussian Elimination Algorithm

Input: A - an augmented $n \times m$ matrix (with a right-hand side)
Output: U - an upper triangular matrix (with a right-hand side)

```
1:  procedure GaussianElimination(A)
2:      now ← 0(目前正在观察的列，寻找主元使用)
3:      for i from 1 to n − 1 do
4:          pst ← −1(主元所在行)
5:          while pst = −1 & now ≠ m do
6:              for j from i to n − 1 do
7:                  if A_{j,now} ≠ 0 then
8:                      pst ← j
9:                      break
10:                 end if
11:                 now ← now + 1 if pst = −1
12:             end for
13:         end while
14:         if pst = −1 then
15:             break
16:         end if
17:         for j from now to m − 1 do
18:             swap(A_{pst,j},a_{i,j})
19:         end for
20:         for j from now + 1 to m − 1 do
21:             A_{i,j} ← A_{i,j} / A_{i,now}
22:         end for
23:         A_{i,now} ← 1
24:         for j from i + 1 to n − 1 do
25:             for k from now + 1 to m − 1 do
26:                 A_{j,k} ← A_{j,k} − A_{i,k} × A_{j,now}
27:             end for
28:             A_{j,now} ← 0
29:         end for
30:         now ← now + 1
31:     end for
32:     return U
33: end procedure
```

为了方便讨论，此处将不讨论更多的特殊情况，数据生成时仅生成 $n \times n$ 的方阵，而且保证选定主元时主元一定出现在下一行。这时，可以去掉寻找主元的过程。由于寻找主元部分时间复杂度为 $O(n^2)$，不贡献主要的时间复杂度，所以这种简化不会影响后续对算法表现的讨论，关于这种数据的生成方式将会在算法设计中讨论。

有了这一简化之后，算法的核心部分只剩两个操作，分别为给当前行除以该行的主元和对后面的行进行消除，使得后续行在当前行主元所在的列处的元素为 0。简化后算法伪代码如下所示。

Algorithm 2 Gaussian Elimination Algorithm

Input: A - an augmented $n \times n$ matrix (with a right-hand side)
Output: U - an upper triangular matrix (with a right-hand side)

```
1: procedure GaussianElimination(A)
2:     for i from 1 to n − 1 do
3:         for j from i + 1 to n − 1 do
4:             A_{i,j} ← A_{i,j} / A_{i,i}
5:         end for
6:         A_{i,i} ← 1
7:         for j from i + 1 to n − 1 do
8:             for k from i + 1 to n − 1 do
9:                 A_{j,k} ← A_{j,k} − A_{i,k} × A_{j,i}
10:            end for
11:            A_{j,i} ← 0
12:        end for
13:    end for
14:    return U
15: end procedure
```

1.2 Grobner 基计算中的高斯消元

1.2.1 Grobner 基理论介绍

理想： 设 $F = \{f_1, \ldots, f_k\}$ 是一个域 K 上的 n 元多项式集合，其生成的理想 $\langle f_1, \ldots, f_k \rangle$ 为集合元素的线性组合，组合系数也是多项式，即

$$\langle f_1, \ldots, f_k \rangle = \left\{ \sum_{i=1}^{k} g_i f_i \;\middle|\; g_1, \ldots, g_k \in K[x_1, \ldots, x_n] \right\}.$$

Grobner 基： 给定一个多项式的理想 I, 多项式集合 G 是 I 的一个 Grobner 基, 当且仅当 G 是最小的包含 I 的理想的生成集之一，且 G 对于 I 中的每个元素执行多元除法的结果均为 0。

首项: 按某种全序关系 (如字典序等) 对各项排列后得到的第一项;
$\mathrm{lp(f)} = X^a$, 即 f 的首项幂积;
$\mathrm{lc(f)} = k$, 即 f 的首项系数。

S-多项式: 给定两个多项式 $f(x_1, x_2, \ldots, x_n)$ 和 $g(x_1, x_2, \ldots, x_n)$,其中 f 和 g 是关于变量 $x_1, x_2, \ldots, x_n$ 的多项式，S-多项式定义为：

$$S(f,g) = \frac{\mathrm{lcm(lp(f), lp(g))}}{\mathrm{lp(f)}} \cdot \mathrm{lc(g)} \cdot f - \frac{\mathrm{lcm(lp(f), lp(g))}}{\mathrm{lp(g)}} \cdot \mathrm{lc(f)} \cdot g$$

其中 lcm 表示对项幂求最小公倍数，即使得 lcm 结果能同时整除两个参与运算的单项式。

Grobner 基的充要条件: G 是一个 Grobner 基，当且仅当多项式集合 G 中任意两个元素的 S-多

项式均对 G 取模为 0。

1.2.2 Grobner 基生成算法

- Buchberger 算法

  Buchberger 算法是一种基于**充要条件判断**的多项式 Grobner 基生成的经典方法。

  1. 初始化：将 $\langle G\rangle$ 初始化为 $\langle H\rangle$;
  2. 通过 S-多项式不断检查 $\langle G\rangle$ 中是否存在某对多项式违背充要条件
     - 若 S-多项式为 0, 未违背，继续下一对检查;
     - 若 S-多项式 s 不为 0, 将 s 添加入 $\langle G\rangle$ 中, 并加入待检查列表。
  3. 重复步骤 2，直到 $\langle G\rangle$ 中任意两个多项式的 S-多项式均检查完成。

  算法伪代码如下

Algorithm 3 Buchberger's Algorithm for Computing Gröbner Bases

Input: F - 一个多项式集合
Output: G - Gröbner 基

```
1:  procedure BUCHBERGER(F)
2:      G ← ∅
3:      for each pair of polynomials p, q ∈ F do
4:          S ← compute S(p, q)
5:          if lc(S) is not divisible by lc(p) and lc(S) is not divisible by lc(q) then
6:              S' ← compute normal form of S with respect to G
7:              if S' ≠ 0 then
8:                  G ← G ∪ {S'}
9:              end if
10:         end if
11:     end for
12:     return G
13: end procedure
```

- F5 算法

  F5 算法通过减少 S-多项式的生成来提高计算效率。F5 算法的核心思想是在每次迭代中，只生成那些不能被当前 Grobner 基约简为零的 S-多项式。这种方法避免了不必要的计算，并且有助于更快地收敛到最终的 Grobner 基。

  1. 初始化一个空集合 G 作为当前的 Grobner 基。
  2. 对于 F 中的每一对多项式 p 和 q，计算它们的 S-多项式 $S(p,q)$。
  3. 对于每个计算出的 S-多项式 $S(p,q)$，如果它的首项不能被 G 中的任何多项式的首项整除，则计算 $S(p,q)$ 在 G 下的标准形式。
  4. 如果 $S(p,q)$ 的标准形式不为零，则将其添加到 G 中。
  5. 重复步骤 2-5，直到没有新的非零 S-多项式可以生成。

  算法伪代码如下

Algorithm 4 F5 Algorithm for Computing Gröbner Bases

Input: F - 一个多项式集合
Output: G - Gröbner 基

```
procedure F5(F)
    G ← ∅
    F' ← F
    while F' ≠ ∅ do
        for each pair of polynomials p, q ∈ F' do
            S ← compute S(p, q)
            if lc(S) is not divisible by any lc(g) for g ∈ G then
                T ← normal form of S with respect to G
                if T ≠ 0 then
                    G ← G ∪ {T}
                    F' ← F' ∪ {T}
                end if
            end if
        end for
        F' ← a set of polynomials in F' whose leading terms are not divisible by any lc(g) for g ∈ G
    end while
    return G
end procedure
```

1.2.3 Grobner 基生成算法中的高斯消元

Grobner 基计算中核心计算部分可以认为是一种特殊的高斯消元，这种高斯消元约束如下。

- 该消元算法执行时，矩阵各行可以被分为两类：消元子和被消元行。
- 运算为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1, 由此，这样的矩阵的行的加减法实际上变为了异或运算。
- 消元子在消元过程中充当 “减数”，被消元行根据首项（首个非零元素的位置）减去消元子，逐步进行消元。所有消元子的首项（可通过将消元子放置在特定行来令该元素位于矩阵对角线上）都不同，但不会涵盖所有对角线元素。如若在消元过程中被消元行不具备被消元的条件（首个非零元素位置没有消元子），此时它被加入到消元子中去，补上此缺失的对角线 1 元素。当所有被消元行都完成消元后，消元完成。

给出伪代码之前，首先解释一些符号含义。

1. R：所有消元子的集合
2. R_i：第一个非零列为第 i 列的消元子，如果不存在则为 $NULL$。
3. E：消元行的集合
4. E_i：第 i 个消元行
5. $lp(E_i)$：E_i 的首项

算法伪代码如下。

Algorithm 5 特殊高斯消元算法

Input: (R,E) - 消元子和被消元行集合
Output: 消元结果

function GAUSS(R, E)
  $R' \leftarrow R$
  $E' \leftarrow E$
  for $i = 1$ to n **do**
    while $E'_i \neq 0$ **do**
      if $R'_{lp(E'_i)} \neq NULL$ **then**
        $E'_i = E'_i - R'_{lp(E'_i)}$
      else
        $R'_{lp(E'_i)} = E'_i$
      end if
    end while
  end for
  return E'
end function

2 普通高斯消元部分

2.1 SIMD 算法设计与复杂度分析

该问题 SIMD 算法的设计非常简单，只需要利用对应的 SIMD 指令将若干次浮点数的运算同时进行即可，由于同一次消去过程中的运算前后顺序可以交换，因此，不需要增加更多的限制。对于多出的部分，长度不足以进行一次并行计算，可以考虑直接串行处理或者补入一些无关数据，由于这部分不是计算的主要时间开销，因此，两种方法的区别不大。算法伪代码如下，这里以 SSE 为例，同时进行 4 个变量的运算，更多变量同时运算与该思想类似。

Algorithm 6 SIMD 优化后的消去部分

Input: 待消元的 $n \times n$ 矩阵 A

function GAUSS(A)
  for k from 0 to $n-1$ **do**
    for j from $k+1$ to $n-1$ **do**
      $A_{k,j} \leftarrow \frac{A_{k,j}}{A_{k,k}}$
    end for
    $A_{k,k} \leftarrow 1$(该部分也可以 SIMD 优化，这里仅以后半部分作为例子)
    for $i = k+1$ to $n-1$ **do**
      $vaik \leftarrow dupTo4Float(A_{i,k})$
      for $i = k+1$ to $n-1$ step= 4 **do**
        $vakj \leftarrow load4Float(\&A_{k,j})$,$vaij \leftarrow load4Float(\&A_{i,j})$
        $vx \leftarrow vakj \times vaik$,$vaij \leftarrow vaij - vx$
        $store4Float(\&A_{i,j}, vaij)$
      end for
      for j in 剩余下标 **do**
        $A_{i,j} = A_{i,j} - A_{i,k} \times A_{k,j}$
      end for
      $A_{i,k} = 0$
    end for
  end for
end function

更进一步的优化，由于一般而言，载入和存储对齐的数据（起始地址为处理长度的倍数）要快于非对齐数据，因此，考虑将前面不对齐的部分串行处理，从地址对齐的位置开始开始使用 SIMD 指令处理，这时，需要保证矩阵每行的起始位置是对齐的，这样，才能实现两行同时对齐。可以通过在堆区申请地址保证矩阵每行起始位置对齐，也可以通过将每行补至对齐后申请连续地址保证对齐。对齐算法伪代码如下（大部分伪代码与前面差别不大，只展部分关键变化）。

Algorithm 7 SIMD 优化后的消去部分

for $i =$ 首个对齐下标 to $n-1$ step= 4 **do**
    $vakj \leftarrow load4Float(\&A_{k,j})$
    $vaij \leftarrow load4Float(\&A_{i,j})$
    $vx \leftarrow vakj \times vaik$
    $vaij \leftarrow vaij - vx$
    $store4Float(\&A_{i,j}, vaij)$
end for
for j in 剩余下标（包括前面未对齐部分和后面不足 4 的部分） **do**
    $A_{i,j} = A_{i,j} - A_{i,k} \times A_{k,j}$
end for
$A_{i,k} = 0$

容易计算，串行算法的时间复杂度为 $T_s = \Theta(n^3)$，而 SIMD 并行几乎没有引入新的计算，完全沿用串行算法，因此，可以认为 SIMD 算法的时间复杂度 $T_p = \Theta(\frac{n^3}{p})$，$E = \frac{S}{p} = \frac{T_s}{p \times T_p} = \Theta(1)$，$cost = p \times T_p = \Theta(n^3)$。因此，该并行方法的可拓展性很好。

2.2 结果展示与分析

最终，实现了串行高斯消元，SSE 不对齐高斯消元，SSE 对齐高斯消元、AVX 不对齐高斯消元、AVX 对齐高斯消元、AVX512 不对齐高斯消元、AVX512 对齐高斯消元、NEON 高斯消元共 8 个普通高斯消元算法。

由于测试环境不同（SSE、AVX、AVX512 在 x86 平台测试，NEON 在 arm 平台测试，选用的不同服务器），所以分为两部分展示测试结果。主要涉及的测试量如下：

- T_s：串行算法运行时间。
- S：加速比，$S = \frac{T_s}{T_p}$，即串行算法运行时间除以并行算法的运行时间。
- E：算法效率，$E = \frac{S}{p} = \frac{T_s}{p \times T_p} = \frac{1}{1 + \frac{T_o}{T_s}}$，即加速比除以线程数量，一般而言，这时一个小于 1 的数字，这是由于并行算法会带来一个额外运行时间 T_o。也存在一些 $E > 1$ 的特殊情况，这可能是受并行算法 cache 访问顺序不同或者节点搜索顺序不同导致的，p 指并行的线程数量，由于 SIMD 编程与一般意义的并行化编程略有区别，并不是通过开辟线程的方法实现并行化。所以，这里按照如下方式解释 p，对于 128 位的 SIMD 指令而言，由于串行算法每次处理的是 32 位数据，而并行算法处理的是 128 位数据，规定 $p = \frac{128}{32} = 4$。

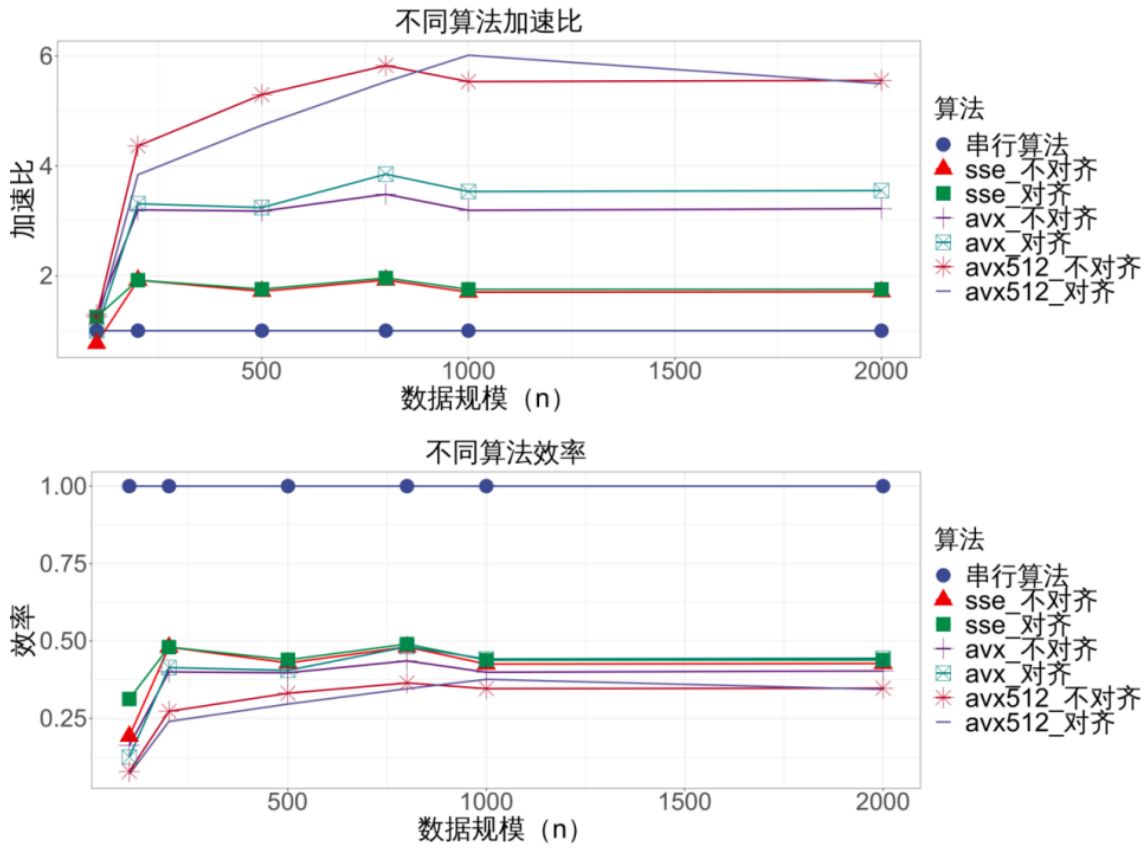
所有测试量都是基于算法运行时间计算的，仅需测量出精确的程序运行时间后进行运算即可得到。测试时间仅包括消元时间，不包括矩阵生成的时间。测试时需测量多次取平均值。

测试时在 6 个不同规模（100，200，500，800，1000，2000）的随机生成矩阵上进行测试。为了保证生成的矩阵在消元过程中不需要寻找主元，采用一种特殊的矩阵生成方式：

1. 生成一个随机的 $n \times n$ 上三角矩阵 A。
2. 对于 i 从 0 到 $n-1$，将 A 的第 i 行加到后面的每一行上。

2.2.1 x86 架构（SSE、AVX、AVX512 编程）

测试结果如下图所示。在不同数据规模下 T_s 的值分别为 1.0ms(n=100)，9.6ms(n=200)，126.1ms(n=500)，599.7ms(n=800)，1005.7ms(n=1000)，8029.2ms(n=2000)。分析数据，可以得到如下结论：



- SIMD 编程的确能够有效地提高算法运行效率，减少运行时间。
- 随着数据规模的增大，算法加速比不断增大，这是由于数据规模的增大，并行带来的额外开销增长速度不如串行算法开销增长速度块，同时，随着数据规模的增大，算法的效率 E 也随之增大，并且，加速比和效率有一个上界，增长到一定程度后就不再增长。
- 随着并行度的增加，算法加速比提高明显，avx512 加速比远大于 avx 的加速比，并且 avx 的加速比远大于 sse 的加速比。这是一个可以预见的结论。
- 随着并行度的增加，算法的效率略微降低，降低并不明显，sse 并行效率略大于 avx，avx 略大于 avx512。这是由于随着并行度的增加，并行带来的额外开销也会随之增加，所以导致效率略有降低，但实际上，这个降低并不明显。一般而言，限制 SIMD 编程的并不是效率降低过快，而是 SIMD 指令集受底层架构影响不支持太多位数据的并行运算。并且，即使在效率上表现最好的并行算法并行效率也远不到 1，这说明并行带来的额外开销是不可忽略的。
- 是否对齐对算法加速比影响不大，对齐算法在加速比和效率上仅仅略高于不对齐算法，甚至有些数据点的表现不如不对齐算法。

2.2.2 arm 架构（NEON 编程）

测试结果如下图所示。在不同数据规模下 T_s 的值分别为 1.9ms(n=100)，15.1ms(n=200)，240.3ms(n=500)，1000.8ms(n=800)，1953.8ms(n=1000)，15839.3ms(n=2000)。

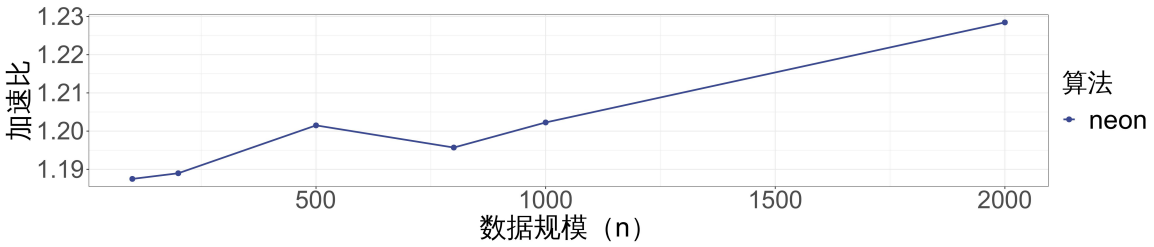


图 2.1: NEON 算法加速比

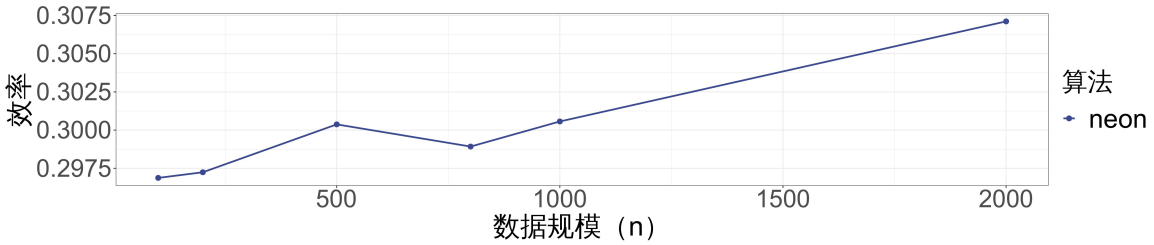


图 2.2: NEON 算法效率

观察数据，可以发现如下规律：

- arm 平台 SIMD 编程也能实现一定加速，并行算法具有一定意义。
- 本次选用的计算机系统 NEON 编程表现不如 SSE 编程，但是由于两者本身并不是在同一实验环境下测定的，所以这种对比参考价值不大。
- 数据规模对加速比和算法效率的影响在 arm 架构的服务器上几乎可以忽略，加速比集中于 1.15-1.25，算法效率集中在 0.295-0.305，对比 x86 架构而言，不同数据规模的差距几乎可以忽略。

3 Grobner 基计算中的高斯消元部分

3.1 SIMD 算法设计与复杂度分析

首先设计如下的串行算法：

1. 初始化消元子为输入的消元子。
2. 逐个待消元行进行消元，如果存在最高位相同的消元子则异或该消元子，否则。
3. 返回消元结果，并且判断是否将待消元行销为 0 向量，如果不是零向量，则将该消元结果加入消元子。

存储数据时有两种方案，第一种思路可以考虑使用位向量存储，也就是每 32 位存储到一个整数中，然后异或时对位异或即可，另一种思路可以考虑使用链表存储，异或时链表合并并减去交集，由

于矩阵比较稀疏，这种方法在某些数据上可能会带来一定优化。本次实验选择第一种方案设计，主要理由有两点，第一，第一种方案表现比较稳定，数据对运行时间的影响较小，体现的是一般性结论，不是基于数据点的结论，第二，这种方法更适合 SIMD 并行化，可能获得更优的加速比。所以采用位向量的方式存储数据。

首先分析该算法时间复杂度，设消元子个数为 n，待消元行个数为 m，每个向量的长度为 N，于是有，共进行 m 次消元，每次消元需要进行异或操作次数不超过 $n+m$（实际上也不会超过 N，但是一般认为 $N >> n+m$），每次异或时间复杂度为 $O(N)$，因此，该算法的时间复杂度为 $O(m(n+m)N)$。当然，由于利用了位向量存储，所以实际上该算法常数很小，实际上位向量存储已经有了一定的 SIMD 思想，每次异或时多位一起进行异或。不过，仍然将时间复杂度表示为 $O(m(n+m)N)$，这里认为这个优化是进行的常数优化。

SIMD 算法的设计仍然不复杂，仅需将贡献主要时间复杂度的运算使用 SIMD 指令并行即可，与普通高斯消元类似，可以设计不对齐算法和对齐算法。

串行算法的时间复杂度为 $T_s = \Theta(m(n+m)N)$，而 SIMD 并行几乎没有引入新的计算，完全沿用串行算法，因此，可以认为 SIMD 算法的时间复杂度 $T_p = \Theta(\frac{m(n+m)N}{p})$，$E = \frac{S}{p} = \frac{T_s}{p \times T_p} = \Theta(1)$，$cost = p \times T_p = \Theta(m(n+m)N)$。因此，该并行方法的可拓展性很好。

3.2 结果展示与分析

与普通高斯消元的测试方式和展示方式类似，但是由于测试点的数据固定，所以不再展示不同规模下的数据，而是展示不同测试点下的数据。由于运行时间较长，仅测试部分具有代表性的数据点，这不会对实验结果造成本质影响。本次实验涉及大量 io 操作，由于本次并行化没有着重考虑优化 io 操作，所以计时时仅计算消元的消耗，可以更好地反映 SIMD 编程的特点。

展示的测试量与普通高斯消元类似，该部分不再介绍展示的测试量的含义和意义。

3.2.1 x86 架构（SSE、AVX、AVX512 编程）

选取的测试点 1：4-1011-539-263，2：6-3799-2759-1953，3：7-8399-6375-4535，4：11-85401-5724-756，选自课程提供的数据。测试数据如下。

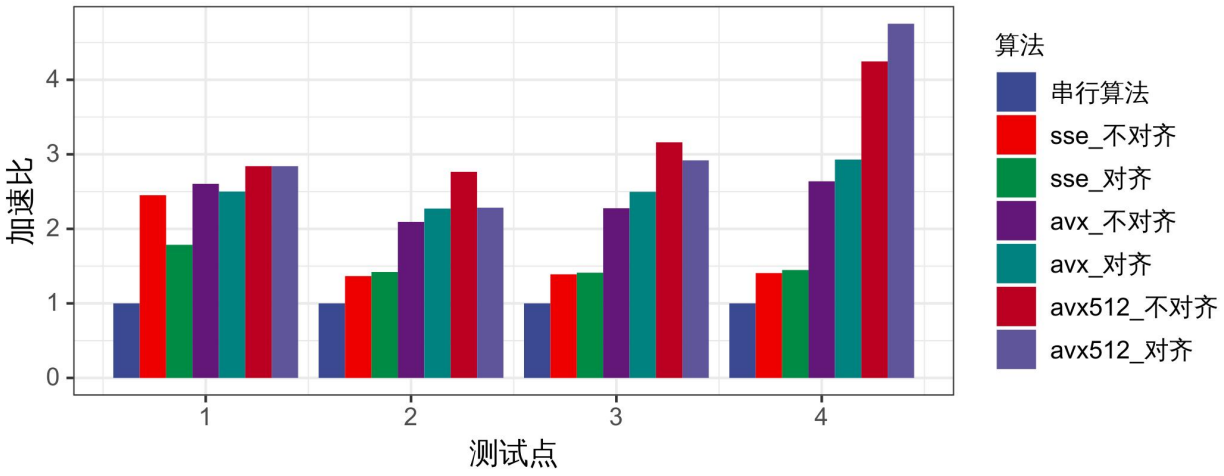


图 3.3: 不同算法加速比

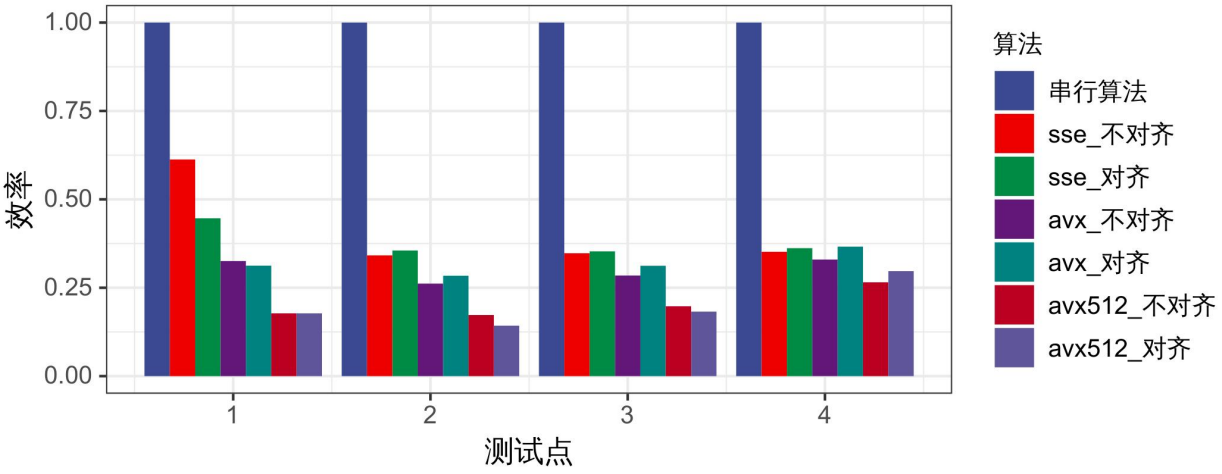


图 3.4: 不同算法效率

观察数据，可以分析出以下结论：

- 与浮点数运算类似，SIMD 编程的确能够有效地提高算法运行效率，减少运行时间。算法在数据规模较大的测试点上表现优于在规模较小的测试点上。随着并行度的增加，算法加速比提高明显。随着并行度的增加，算法的效率略微降低，降低并不明显。并且，数据规模越大，这个降低越不显著。是否对齐对算法加速比影响不大，对齐算法在加速比和效率上仅仅略高于不对齐算法，甚至有些数据点的表现不如不对齐算法。
- 不论哪种算法，对整数异或运算的加速比都不如对浮点数运算的加速比表现好，这可能是由于硬件对浮点数 SIMD 编程做了一定优化。浮点数运算最优可以达到 5.5 左右的加速比，而整数运算只能达到 4.7 左右，对应的非最优算法的加速比浮点数运算也要比整数运算更高。

3.2.2 arm 架构（NEON 编程）

测试数据如下。

测试点	1	2	3	4
加速比S	1.074	1.107	1.087	1.108
效率E	0.268	0.277	0.272	0.277

可以发现，不论是相较 SSE 编程，还是相较 NEON 的浮点数运算，NEON 整数异或运算得到的加速都很小，加速比在 1.1 左右，这个优化远不如前面的优化效果明显。当然，这并不意味着这个优化毫无意义，在硬件资源大量空闲的情况下，可以接受并行效率不高的情况下，仍然可以采用这个优化。

4 实验环境与仓库链接

4.1 实验环境

环境 1（x86 环境）

- 处理器 Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- 操作系统 Ubuntu 20.04.6 LTS
- 编译器 g++ (Ubuntu 9.4.0-1ubuntu1 20.04.2) 9.4.0
- 编译选项 g++ % -o %< -msse/msse2/mavx/mavx2/mavx512f -O0 -fpermissive -std=c++11

环境 2（arm 环境）

- 处理器/操作系统课程提供的鲲鹏服务器处理器/操作系统
- 编译器 g++ (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2)
- 编译选项 g++ % -o %< -march=armv8-a -O0 -fpermissive -std=c++11

4.2 仓库链接

https://github.com/pler1010/para，该仓库 main 分支下，SIMD 文件夹下是本次实验的程序。