南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程实验

姓名：陆遥

学号：2211843

专业：计算机科学与技术

2024 年 4 月 24 日

目录

1 问题描述

数学上，高斯消元法（或译：高斯消去法），是线性代数规划中的一个算法，可用来为线性方程组求解。但其算法十分复杂，不常用于加减消元法，求出矩阵的秩，以及求出可逆方阵的逆矩阵。不过，如果有过百万条等式时，这个算法会十分省时。一些极大的方程组通常会用迭代法以及花式消元来解决。当用于一个矩阵时，高斯消元法会产生出一个“行梯阵式”。高斯消元法可以用在电脑中来解决数千条等式及未知数。亦有一些方法特地用来解决一些有特别排列的系数的方程组。

2 普通高斯消去算法

2.1 算法设计

2.1.1 传统串行算法

高斯消去的计算模式如图 2.1 所示，主要分为消去过程和回代过程。在消去过程中进行第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作，全部结束后，得到如图 2.2 所示的结果。而回代过程从矩阵的最后一行开始向上回代，对于第 i 行，利用已知的 xi+1, xi+2, ..., xn 计算出 xi。串行算法如下面伪代码所示。

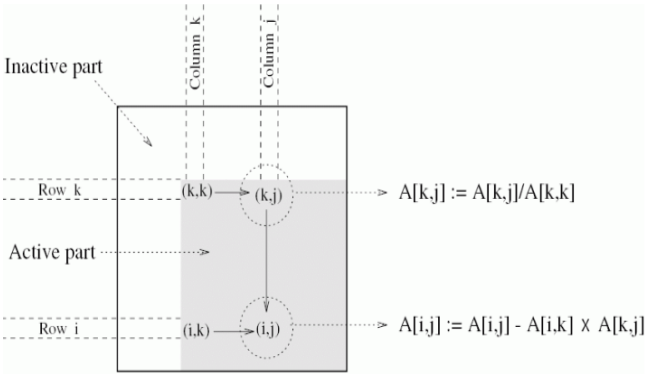

图 2.1: 高斯消去法示意图

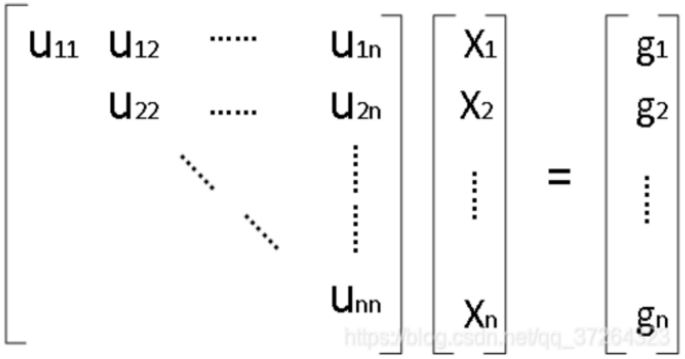

图 2.2: 高斯消去法消去过程结束后，结果的示意图

2.1.2 SSE 编程并行优化算法

下面给出一个使用 SIMD Intrinsics 函数对普通高斯消元进行向量化的伪代码，见算法 1，基本上可以逐句翻译为 Neon 高斯消元函数。这里只给出了支持内存不对齐的 SIMD 访存操作。在 x86 平台上还可以考虑是否使用了支持内存不对齐的访存指令，使用内存对齐的访存指令需要对起始下标进行调整。

Algorithm 1: 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 ← dupTo4Float(A[k,k]);
    for j = k + 1; j + 4 <= n; j+ = 4 do
        va ← load4FloatFrom(&A[k,j]) ;      // 将四个单精度浮点数从内存加载到向量寄存器
        va ← va/vt ;                                          // 这里是向量对位相除
        store4FloatTo(&A[k,j],va);          // 将四个单精度浮点数从向量寄存器存储到内存
    for j in 剩余所有下标 do
        A[k,j]=A[k,j]/A[k,k] ;                    // 该行结尾处有几个元素还未计算
    A[k,k] ← 1.0;
    for i ← k+1 to n-1 do
        vaik ← dupToVector4(A[i,k]);
        for j = k + 1; j + 4 <= n; j+ = 4 do
            vakj ← load4FloatFrom(&A[k,j]);
            vaij ← load4FloatFrom(&A[i,j]);
            vx ← vakj*vaik;
            vaij ← vaij-vx;
            store4FloatTo(&A[i,j],vaij);
        for j in 剩余所有下标 do
            A[i,j] ← A[i,j] -A[k,j]*A[i,k];
        A[i,k] ← 0;
```

图 2.3: SIMD 编程并行优化算法

2.2 对比实验设计与数据处理

2.2.1 内存对齐与不对齐进行对比实验

设计对齐与不对齐算法策略时，注意到高斯消去计算过程中，第 k 步消去的起始元素 k 是变化的，从而导致距 16 字节边界的偏移是变化的。

对于 x86 平台的实验，如果设计对齐算法时，可以调整算法，先串行处理到对齐边界，然后进行 SIMD 的计算。可对比两种方法的性能。C++ 中数组的初始地址一般为 16 字节对齐，所以只要确保每次加载数据 A[i : i + 3] 中 i 为 4 的倍数即可。

2.2.2 对不同部分的优化进行对比实验

高斯消去法中有两个部分可以进行向量化，我们可以对比一下这两个部分 (一个二重循环、一个三重循环) 进行 SIMD 优化对程序速度的影响。

2.2.3 并行计算结果的误差处理

并行计算由于重排了指令执行顺序，加上计算机表示浮点数是有误差的，可能导致即使数学上看是完全等价的，但并行计算结果与串行计算结果不一致。这不是算法问题，而是计算机表示、计算浮点数的误差导致，一种策略是允许一定误差，比如 $< 10e{-}6$ 就行；另外一种策略，可在程序中加入一些数学上的处理，在运算过程中进行调整，来减小误差。

2.3 代码实现

篇幅所限，部分函数仅展示部分代码，省略与其他函数重复部分。

2.3.1 传统串行算法

```
void serial(int n)//传统串行算法
{
    for(int k=0;k<n;k++)
    {
        for(int j = k+1 ; j < n ; j++)
        {
            A[k][j] = A[k][j]/A[k][k];
        }
        A[k][k] = 1.0;
        for(int i = k+1 ; i < n ; i++)
        {
            for(int j = k+1 ; j < n ; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

2.3.2 乘法部分向量化，不对齐

```
void SSE_1(int n)//乘法部分向量化，不对齐
{
     __m128 factor4 = _mm_set_ps1(A[i][k]);
    int j;
    for(j=k+1;j+4<=n;j+=4)
    {
        __m128 vaij = _mm_loadu_ps(&A[i][j]);
```

```
        __m128 vakj = _mm_loadu_ps(&A[k][j]);
        vakj =  _mm_mul_ps(vakj,factor4);
        vaij = _mm_sub_ps(vaij,vakj);
        _mm_store_ps(&A[i][j],vaij);
    }
}
```

2.3.3 除法部分向量化，不对齐

```
void SSE_2(int n)//除法部分向量化，不对齐
{
    int j;
    __m128 vt = _mm_set1_ps(A[k][k]);
    for(j = k+1 ; j+4 <= n ; j+=4)
    {
        __m128 va = _mm_loadu_ps(&A[k][j]);
        va = _mm_div_ps(va,vt);
        _mm_store_ps(&A[k][j],va);
    }
}
```

2.3.4 乘法、除法都向量化，不对齐

结合上面乘法部分向量化，不对齐和除法部分向量化，不对齐代码。

2.3.5 乘法部分向量化，对齐

```
void SSE_4(int n)//乘法部分向量化，对齐
{
    int start = k+4-k%4;
    for(j=k+1;j<start && j<n;j++)
    {
        A[i][j] = A[i][j] - A[k][j]*A[i][k];
    }
}
```

2.3.6 除法部分向量化，对齐

```
void SSE_5(int n)//除法部分向量化，对齐
{
    int start = k+4-k%4;
    for(j=k+1;j<start && j<n;j++)
    {
        A[k][j] = A[k][j]/A[k][k];
    }
```

```
}
```

2.3.7 乘法、除法都向量化，对齐

结合上面乘法部分向量化，对齐和除法部分向量化，对齐代码。

2.4 实验结果

2.4.1 实验平台运行

在 Code::Blocks 中运行实验程序。

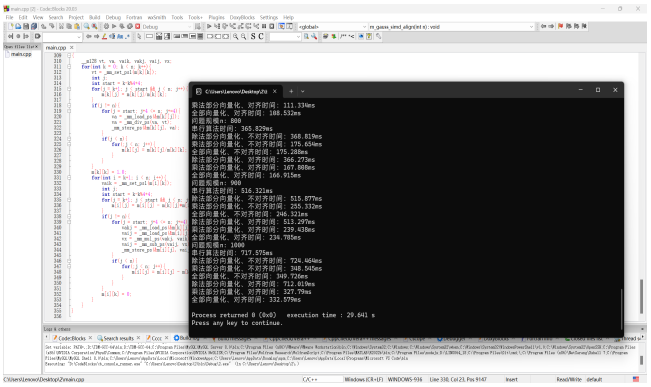

图 2.4: 实验平台运行图

2.4.2 不同算法和问题规模下运行用时表

单位：ms

问题规模	n=10	n=50	n=80	n=100	n=500	n=800	n=1000
传统串行算法	0.0014	0.1005	0.4328	1.04	86.6804	361.517	716.355
除法部分向量化、不对齐	0.003	0.101	0.4256	0.795	86.1064	371.952	712.763
乘法部分向量化、不对齐	0.0014	0.0503	0.3144	0.4	38.6015	186.241	345.781
全部向量化、不对齐	0.0014	0.051	0.2895	0.4615	38.6583	175.18	338.889
除法部分向量化、对齐	0.0017	0.1266	0.4364	1.0965	87.1214	367.267	713.501
乘法部分向量化、对齐	0.0054	0.0634	0.2052	0.3373	36.4085	171.438	323.065
全部向量化、对齐	0.0019	0.1058	0.2036	0.3547	36.9403	167.303	314.683ms

2.4.3 实验结果总结

1、问题规模的影响：问题规模较小时，总体并行优化算法相较于传统串行用时相似，除法部分向量化算法甚至出现用时增长的情况。随着问题规模的增大，并行算法的优化效果才开始显现。

2、不同部分向量化的影响：对除法部分向量化优化效果不加，只在问题规模到达 1000 后才略有效果，而对乘法部分向量化效果明显，总体的向量化带来的用时减少主要来源于乘法部分向量化的效果。

3、内存对齐的影响：问题规模较小时，内存对齐起到了反作用，但是随着问题规模的增大，内存对齐的优化效果开始显现。

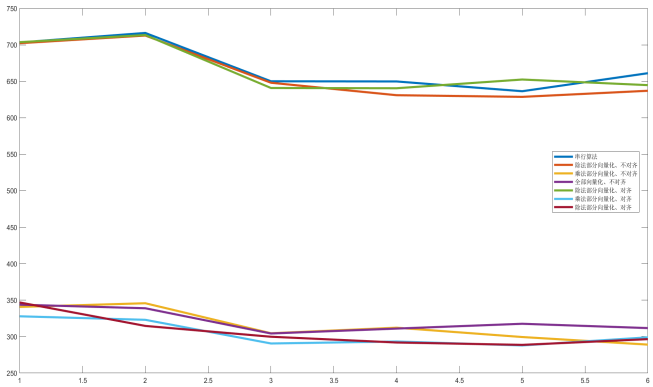

图 2.5: 问题规模为 1000，不同算法下运行用时折线图

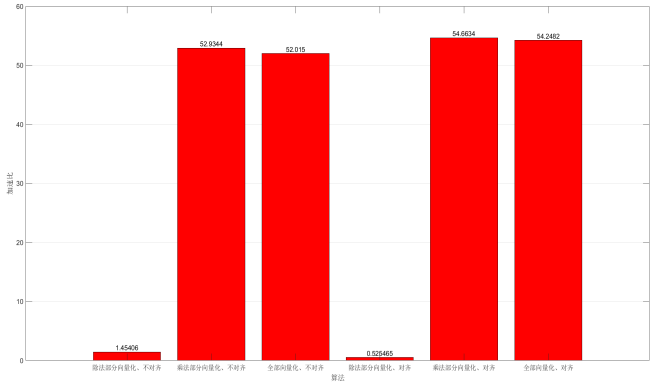

图 2.6: 不同算法平均加速比柱状图

2.5 性能分析

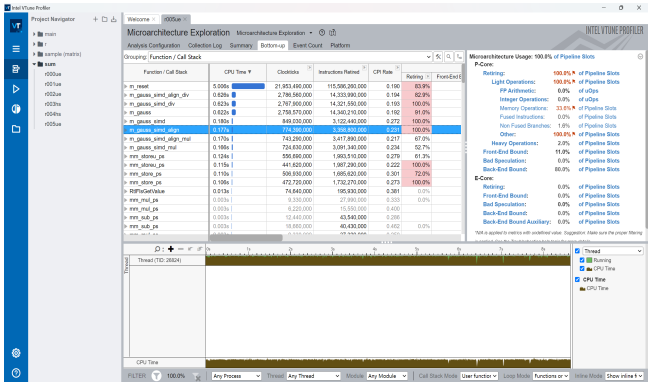

图 2.7: VTune Proflier 分析图

对问题规模 n=1000 时各算法的性能进行具体的分析。

我们主要看的是 Instructions Retired 和 CPI Rate 这两个指标，第一个是函数执行的总指令数，第二个是平均执行一条指令所需的时钟周期数。

具体分析来看，由于 SIMD 并行化编程使用了多路向量寄存器，可以同时使多个数进行加减等运算，因此其执行的总指令数相对较少，但是指令操作较为复杂，平均执行一条指令所需时间较长，这两种因素分别会延长和缩减函数执行时间。但总体来看，虽然 SIMD 并行化使 CPI 增大，但是其并没有与多路计算的路数呈相应的倍数关系，而是更小，因此总的函数时间会缩短，性能得到优化。

2.6 与 ARM 架构下实验对比

为了能在 ARM 架构下测试，需要在华为鲲鹏服务器编译与运行程序，在真正 ARM 机器上运行，实验结果是 ARM SIMD 真实性能。

需要重写代码，以适应 ARM 架构下的运行环境，在本次实验中，使用了 Neon 指令集，在 arm-neon 头文件下进行多路数据的向量化运算，但是整体指令操作逻辑完全相同。

展示部分代码

```
void Neon(int n)
{
    float32x4_t vaij = vld1q_f32(&A[i][j]);
    float32x4_t vakj = vld1q_f32(&A[k][j]);
    vakj = vmulq_f32(vakj,factor4);
    vaij = vsubq_f32(vaij,vakj);
    vst1q_f32(&A[i][j],vaij);
}
```

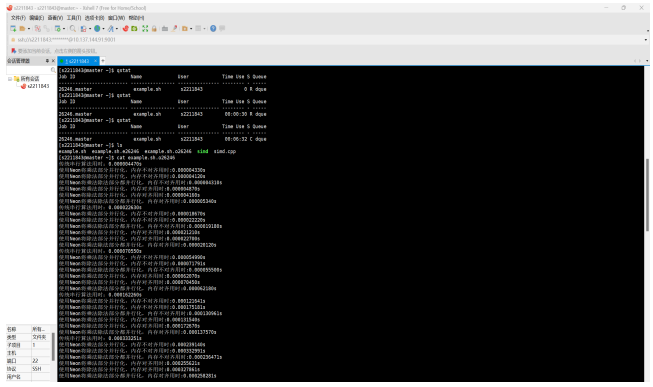

图 2.8: 鲲鹏服务器运行图

问题规模	n=10	n=50	n=100	n=200	n=500	n=1000
传统串行算法	0.004470	0.33325	2.4824	19.887	310.242	2469.019
除法并行化，不对齐	0.004120	0.33329	2.4610	19.406	310.989	2473.099
乘法并行化，不对齐	0.004330	0.23914	1.6931	13.115	203.936	1626.423
全部并行化，不对齐	0.004310	0.23647	1.6940	13.117	203.776	1621.822
除法并行化，对齐	0.004160	0.32786	2.4781	19.663	310.504	2482.081
乘法并行化，对齐	0.004870	0.25562	1.7431	13.477	209.630	1683.592
全部并行化，对齐	0.005340	0.25858	1.7452	13.521	209.077	1677.690

对比分析 总体来讲，ARM 架构下呈现运行时间普遍变长，但是并行化算法的优化效果大致相同。

也有一定的区别，主要体现为：内存对齐的优化效果在问题规模 n=1000 时仍然没有得到体现，推测在问题规模更大的情况下才能够提高性能。

3 特殊高斯消去计算

3.1 算法介绍

特殊高斯消去计算来自一个实际的密码学问题–Gröbner 基的计算与普通高斯消去计算的区别如下:

1、运算均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。1. 其加法运算实际上为异或运算:0+0=0、0+1=1、1+0=1、1+1=0 由于异或运算的逆运算为自身，因此减法也是异或运算。乘法运算 0*0=0、0*1=0、1*0=0、1*1=0。因此，高斯消去过程中实际上只有异或运算——从一行中消去另一行的运算退化为减法。

2、矩阵行分为两类，“消元子” 和 “被消元行”，在输入时即给定。消元子是在消去过程中充当 “减数” 的行，不会充当 “被减数”。所有消元子的首个非零元素 (即首个 1，称为首项) 的位置 (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。被消元行在消去过程中充当 “被减数”，但有可能恰好包含消元子中缺失的对角线 1 元素，此时它 “升格” 为消元子，补上此缺失的对角线 1 元素。

3.2 算法设计

3.2.1 传统串行算法

a) 实际问题中矩阵规模很大，消元子和被消元行的数量很多 (可能达到百万级)，大大超出内存容量，一种处理方式是逐批次将消元子和被消元行读入内存，执行下面步骤 b)-c);

b) 对当前批次中每个被消元行，检查其首项，如有对应消元子则将其减去 (异或) 对应消元子，重复此过程直至其变为空行 (全 0 向量) 或首项不在当前批次覆盖范围内、或首项在范围内但无对应消元子或该行，若为情况 2 则该行此批次计算完成;

c) 如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算; 如其首项被当前批次覆盖，但没有对应消元子，则将它 “升格” 为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与; 重复上述过程，直至所有批次都处理完毕，此时消元子和被消元行共同组成结果矩阵——可能存在很多空行，

3.2.2 SSE 编程并行优化算法

和普通高斯消去算法类似，重点在循环处进行多路向量化并行处理。值得一提的是，在设计此 SSE 编程并行优化算法时不再考虑内存对齐的问题，但是会对比不同路向量化对计算的影响。

3.3 代码实现

为了实现特殊高斯消去计算，程序中有多个辅助函数，但是由于篇幅有限，因此不作展示。

3.3.1 传统串行算法

```
1 void serial()
2 {
3     int begin = 0;
4     int flag;
5     flag = readRowsFrom(begin);       //读取被消元行
```

```
    int num = (flag == -1) ? maxrow : flag;
    for (int i = 0; i < num; i++) {
        while (findfirst(i)!= -1) {      //存在首项
      int first =findfirst(i);       //first是首项
      if (ifBasis[first]==1) {  //存在首项为first消元子
        for (int j = 0; j < maxsize; j++) {
            gRows[i][j] = gRows[i][j] ^ gBasis[first][j];      //进行异或消元
        }
            }
      else {    //升级为消元子
        for (int j = 0; j < maxsize; j++) {
          gBasis[first][j] = gRows[i][j];
        }
        //iToBasis.insert(pair<int, int*>(first, gBasis[first]));
        ifBasis[first] = 1;
        ans.insert(pair<int, int*>(first, gBasis[first]));
        break;
      }
    }
    }
}
```

3.3.2 4 路向量化算法

```
void SSE_4()
{
    int j = 0;
    for (; j + 4 <= maxsize; j += 4) {
    __m128i vij = _mm_loadu_si128((__m128i*) &gRows[i][j]);
    __m128i vj = _mm_loadu_si128((__m128i*) &gBasis[first][j]);
    __m128i vx = _mm_xor_si128(vij, vj);
    _mm_store_si128((__m128i*) &gRows[i][j], vx);
    }
}
```

3.3.3 8 路向量化算法

```
void SSE_8()
{
    int j = 0;
    for (; j + 8 <= maxsize; j += 8) {
    __m256i vij = _mm256_loadu_si256((__m256i*) &gRows[i][j]);
    _mm256_store_si256((__m256i*) &gBasis[first][j], vij);
    }
}
```

3.4 实验结果

3.4.1 实验平台运行

在 Code::Blocks 中运行实验程序。

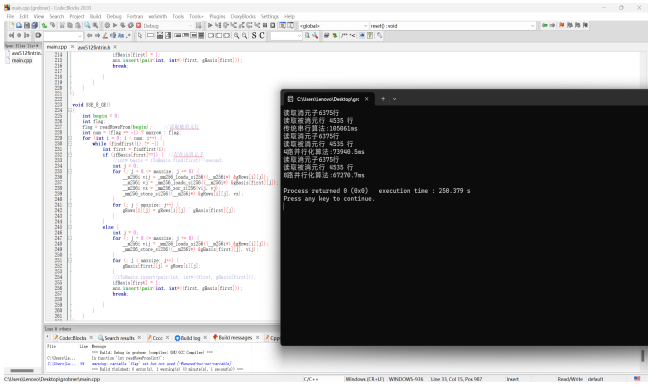

图 3.9: 实验平台运行图

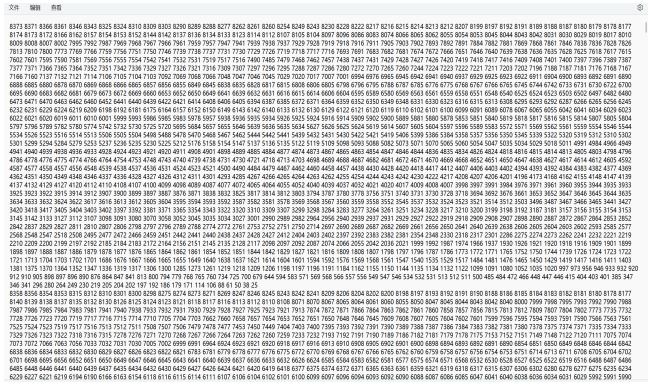

图 3.10: 消元结果

3.4.2 不同算法和测试样本下运行用时表

单位：ms

测试样本编号	1	2	3	4	5	6	7
传统串行算法	44.8257	60.9992	60.1359	428.298	1560.56	17327.3	105061
4 路向量化	44.7444	56.3106	56.8382	317.522	1094.77	12545.6	73940.5
8 路向量化	45.591	54.7484	62.5499	292.593	985.426	11397.5	67270.7

3.4.3 实验结果总结

随着测试样例编号的增长，问题规模也在增长。开始时优化效果并不明显，甚至出现时间增长的现象，但是后来优化效果越来越明显，且 8 路向量化的优化效果好于 4 路向量化。

4 链接

github 项目链接:https://github.com/TimeIsAPlace/SIMD.git