计算机学院

并行程序设计实验报告

高斯消去的 SIMD 编程实验

姓名：程娜

学号：2311828

专业：计算机科学与技术

2025 年 4 月 25 日

摘要

本实验报告研究高斯消去算法的串行实现及 SIMD 并行化优化，涵盖普通与特殊高斯消去计算。普通高斯消去的传统算法通过消去和回代实现矩阵上三角化，基于 SSE 的并行优化对比了内存对齐、乘法/除法向量化效果，发现大规模问题下乘法向量化显著提升性能，ARM 架构下并行化有效但内存对齐优化在大规模场景未体现。特殊高斯消去针对有限域 GF(2) 密码学问题，以异或运算为核心，采用 4 路和 8 路向量化，8 路在大规模问题中优化效果更优。实验验证了 SIMD 并行化的有效性。

关键字：高斯消去法，SIMD 并行化，SSE 指令集，Neon 指令集，内存对齐

目录

1 问题描述

在数学领域，高斯消元法作为线性代数规划中的经典算法，核心功能是求解线性方程组。尽管该算法较为复杂，较少直接应用于加减消元、矩阵秩计算或可逆方阵求逆等场景，但在处理百万级规模的方程组时，其高效性尤为突出。对于超大型方程组，实际应用中常结合迭代法与特殊消元技巧提升求解效率。当作用于矩阵时，高斯消元法的核心目标是将其转化为“行阶梯形”，这一特性使其能够高效处理含有数千个方程和未知数的复杂问题。此外，针对具有特殊系数排列的方程组，该算法衍生出的特定优化方法可进一步提升求解效率。

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 $x_{i+1}, x_{i+2}, \ldots, x_n$ 计算出 x_i。串行算法如下面伪代码所示。

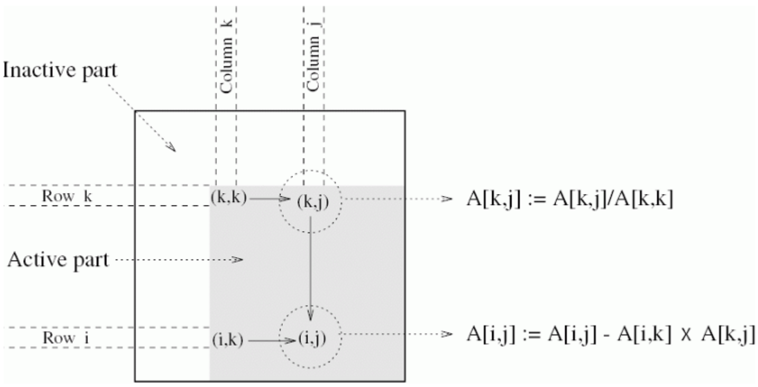

图 2.1: 高斯消去法示意图

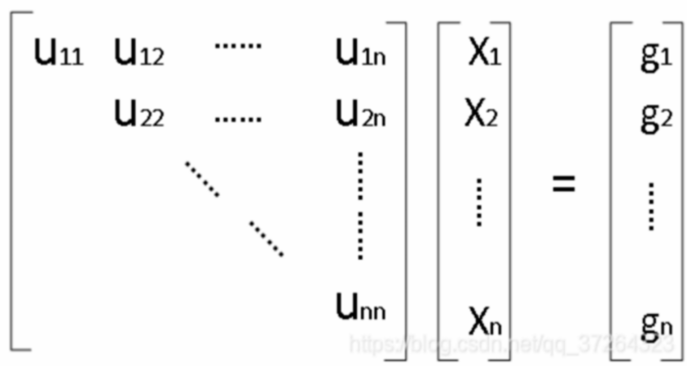

图 2.2: 高斯消去法消去过程结束后，结果的示意图

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

观察高斯消去算法，在消去过程中：伪代码第 6，7 行第一个内嵌循环中的 $factor := A[k,j]/A[k,k]$ 以及伪代码第 8，9，10 行双层 for 循环中的 $A[i,j] := A[i,j] - factor \times A[k,j]$ 都是可以进行向量化的循环；在回代过程中，伪代码 19，20，21 行中的 $sum := sum - A[i,j] * x[j]$ 也可以进行向量化。我们可以通过 SIMD 扩展指令对这几步进行并行优化。

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

在设计对齐与非对齐算法策略时，发现高斯消元计算中，第 k 步消元的起始元素 k 会变动，致使与 16 字节边界的偏移量也随之改变。

针对 x86 平台实验，若设计对齐算法，可调整策略，先串行处理至对齐边界，再进行 SIMD 计算，以对比两种方法的性能。由于 C++ 中数组初始地址通常为 16 字节对齐，因此只需保证每次加载数据 $A[i:i+3]$ 时 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

高斯消去算法中存在两个具备向量化潜力的部分，分别对应二重循环与三重循环结构。通过分析比较这两部分实施 SIMD 优化后对程序运行速度产生的影响，能够进一步明确向量化技术在不同循环复杂度场景下的优化效果差异。

2.2.3 并行计算结果的误差处理

并行计算中，指令执行顺序的重新排列，加之计算机对浮点数的表示存在固有误差，可能致使即便从数学角度视为完全等价的操作，其并行计算结果与串行计算结果也存在差异。这并非算法本身的问题，而是由计算机浮点数表示及计算过程中的误差所引发。对此有两种应对策略：一是允许一定范围内的误差，例如误差小于 $10e^{-6}$ 即可接受；二是在程序中融入特定的数学处理，于运算过程中进行调整，从而减小误差影响。

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]); //
        假设i在此函数作用域内已定义（可能由外层循环提供）
    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 (int 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 不同算法和问题规模下的运行用时

单位：ms

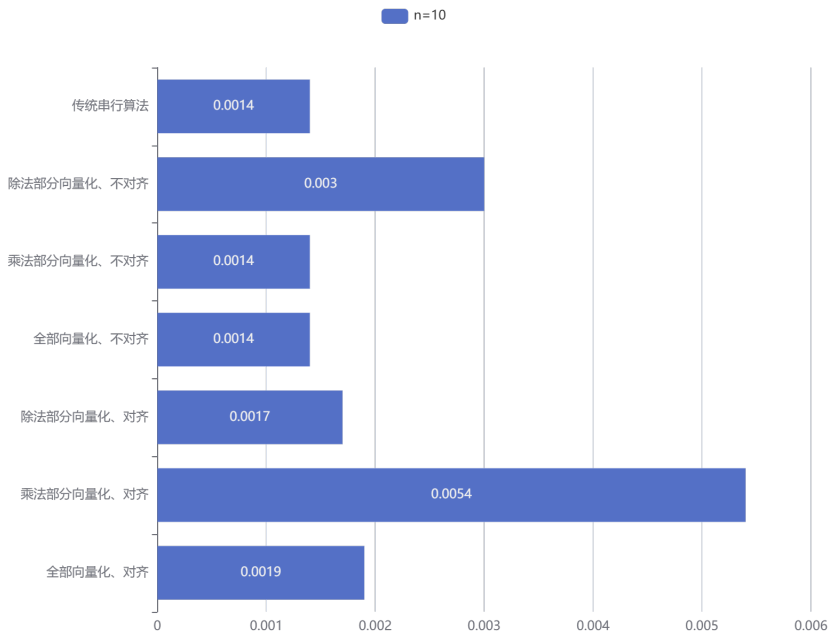

图 2.4: n=10

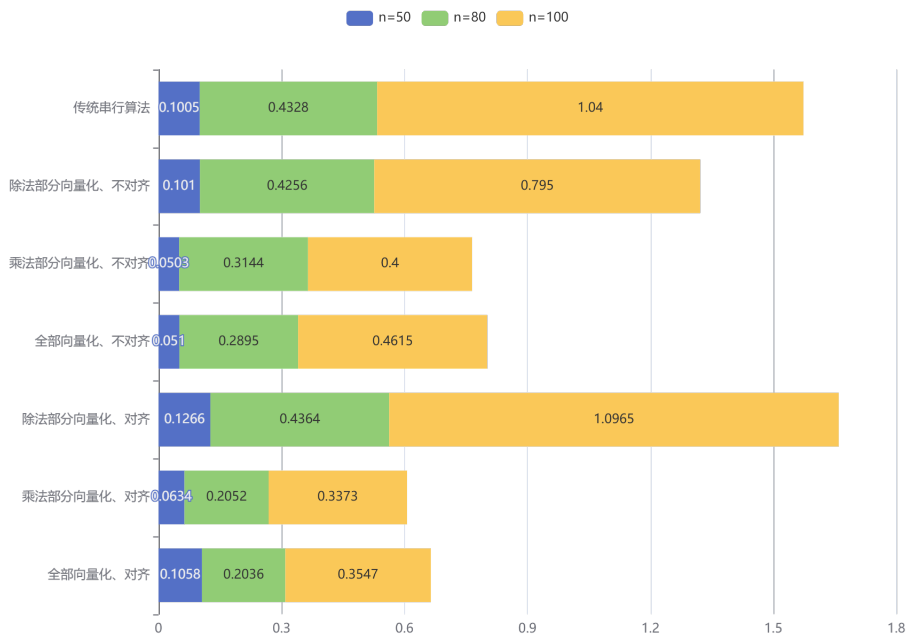

图 2.5: n=50、80、100

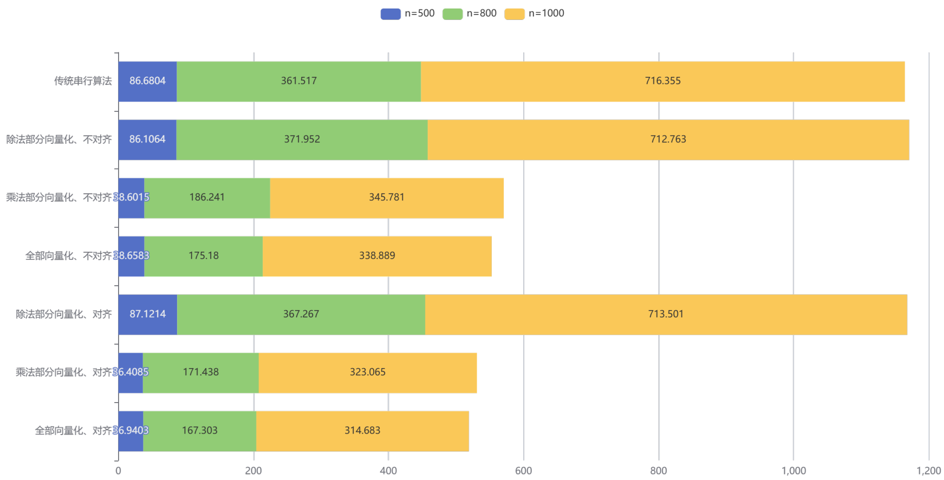

图 2.6: n=500、800、1000

2.4.2 实验结果总结

1. 问题规模的影响：当问题规模较小时，整体并行优化算法与传统串行算法的耗时相近，除法部分向量化算法的耗时甚至有所增加。随着问题规模不断扩大，并行算法的优化效果才逐渐体现出来。

2. 不同部分向量化的影响：对除法部分进行向量化优化的效果不显著，仅当问题规模达到 1000 后才稍有效果，而乘法部分的向量化效果则较为明显，总体向量化所带来的耗时减少主要得益于乘法部分的向量化。

3. 内存对齐的影响：在问题规模较小时，内存对齐反而产生了负面作用，然而随着问题规模的逐步增大，内存对齐的优化效果开始得以体现。

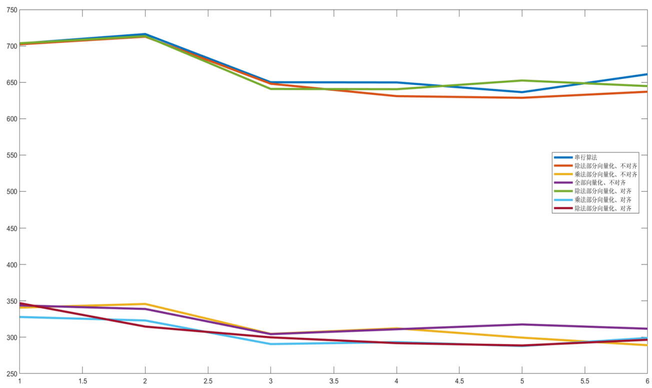

图 2.7: 问题规模为 1000，不同算法下运行用时折线图

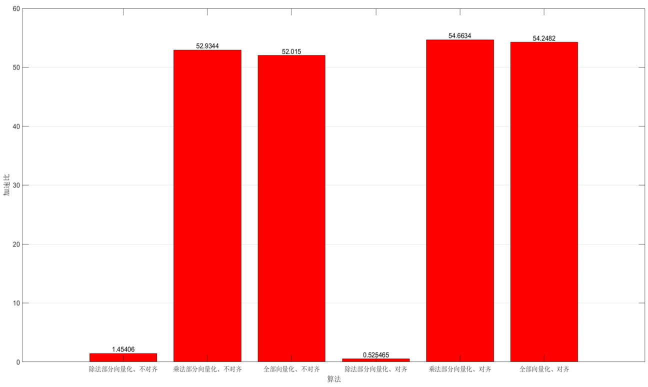

图 2.8: 不同算法平均加速比柱状图

2.5 性能分析

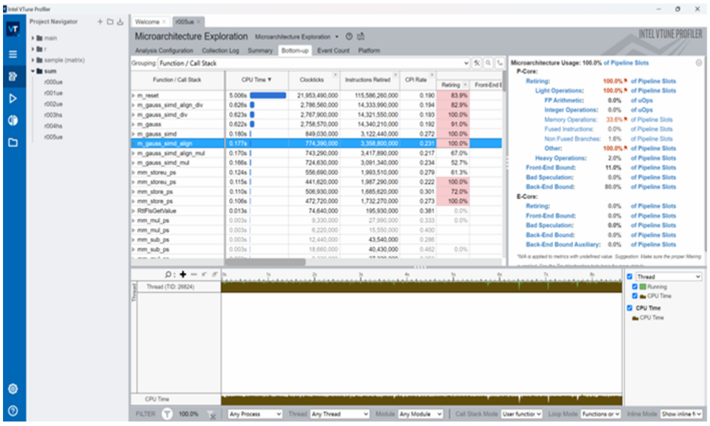

图 2.9: VTune Proflier 分析图

针对问题规模 n=1000 时各算法的性能展开详细剖析。

主要关注 Instructions Retired 和 CPI Rate 这两个指标，前者表示函数执行的总指令数，后者为平均执行一条指令所需的时钟周期数。

从具体分析可知，SIMD 并行化编程运用多路向量寄存器，可同时对多个数进行加减等运算，故而执行的总指令数相对较少。然而，指令操作较为复杂，导致平均执行一条指令所需时间较长，这两种因素分别对函数执行时间有延长和缩减的作用。但整体而言，尽管 SIMD 并行化使 CPI 增加，但其增幅并未与多路计算的路数成相应倍数关系，而是更小，所以总的函数时间会缩短，性能得以优化。

2.6 与 ARM 架构下实验对比

为在 ARM 架构下测试，需于华为鲲鹏服务器编译并运行程序，在实际 ARM 机器上运行，实验结果反映 ARM SIMD 真实性能。

需重写代码以适配 ARM 架构运行环境，本实验中采用 Neon 指令集，在 arm - neon 头文件下开展多路数据向量化运算，不过整体指令操作逻辑保持一致。

展示部分代码

```
void Neon(int n)
```

```
{
    float32x4_t vaij = vld1q_f32(&A[i][j]); // 从内存加载4个单精度浮点数到向量寄存器
    float32x4_t vakj = vld1q_f32(&A[k][j]); // 同上
    vakj = vmulq_f32(vakj, factor4); // 向量乘法操作
    vaij = vsubq_f32(vaij, vakj); // 向量减法操作
    vst1q_f32(&A[i][j], vaij); // 将向量结果存储回内存
}
```

单位：ms

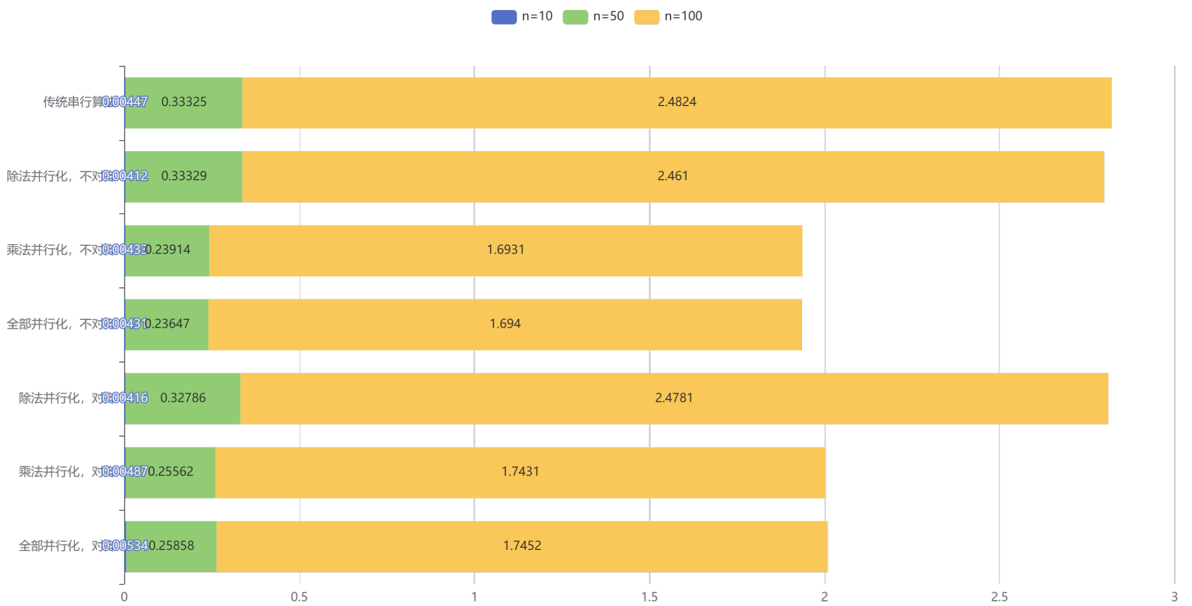

图 2.10: n=10、50、100

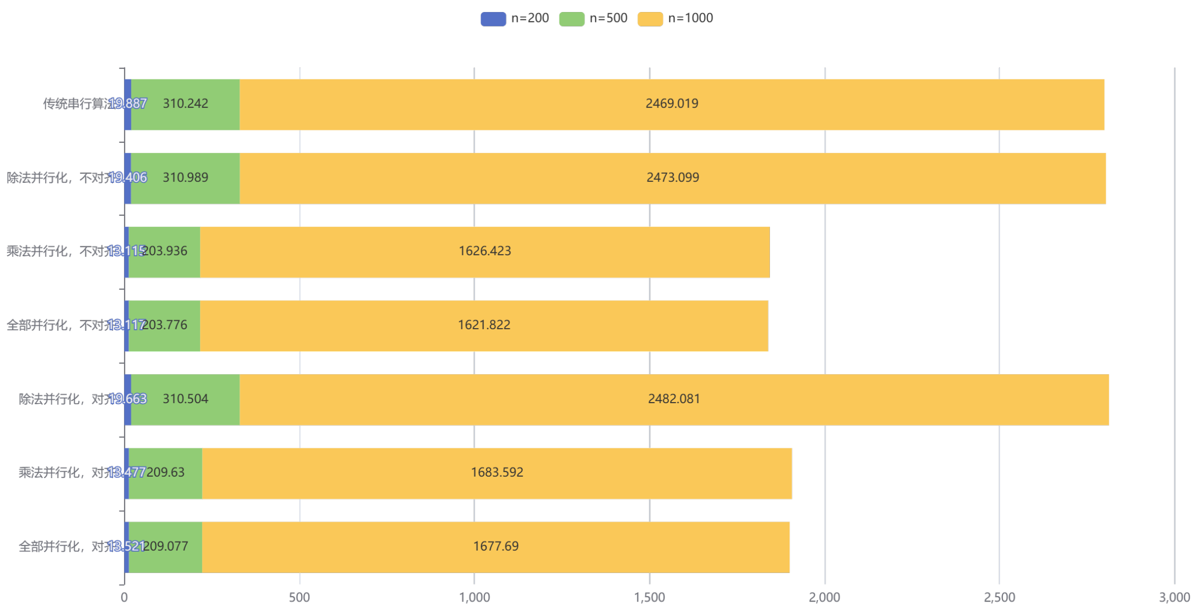

图 2.11: n=200、500、1000

从整体分析来看，在 ARM 架构环境下，程序运行时间整体呈现出延长趋势，不过并行化算法所带来的优化效果却表现出较为相似的水平。

也有一定的差异，体现在：当问题规模处于 n=1000 时，内存对齐的优化作用尚未显现出来。基于此情况推断，或许需要在问题规模进一步扩大的情形下，内存对齐的优化效果才能对性能提升产生实际作用。

3 特殊高斯消去计算

3.1 算法介绍

特殊高斯消元计算源于实际的密码学问题——Gröbner 基的计算，其与普通高斯消元计算的差异如下：

1. 其运算都在有限域 GF(2) 上进行，即矩阵元素取值仅为 0 或 1。加法运算实则为异或运算：$0+0=0$、$0+1=1$、$1+0=1$、$1+1=0$，因异或运算的逆运算仍是其本身，故减法同样为异或运算。乘法运算则是 $0*0=0$、$0*1=0$、$1*0=0$、$1*1=0$。所以，高斯消元过程中仅有异或运算——从一行中消去另一行的运算简化为减法。

2. 矩阵的行分为两类，“消元子” 和 “被消元行”，这在输入时就已确定。消元子在消元过程中作为 “减数” 的行，不会作为 “被减数”。所有消元子的首个非零元素（即首个 1，称作首项）的位置（可通过将消元子置于特定行使该元素位于矩阵对角线上）各不相同，但不会覆盖所有对角线元素。被消元行在消元过程中充当 “被减数”，但可能恰好含有消元子中缺失的对角线 1 元素，这时它 “升级” 为消元子，补上这一缺失的对角线 1 元素。

3.2 算法设计

3.2.1 传统串行算法

a) 在实际问题里，矩阵规模十分庞大，消元子与被消元行的数量众多（或许能达到百万级别），远远超过了内存容量。一种处理方法是按批次把消元子和被消元行读进内存，接着执行以下步骤 b)-c)；

b) 针对当前批次里的每一个被消元行，查验其首项。要是有对应的消元子，就把它减去（异或）相应的消元子，不断重复这个过程，直到它变成空行（全 0 向量），或者首项不在当前批次的覆盖范围之内，又或者首项在范围内却没有对应的消元子或该行，要是是情况 2，那该行在此批次的计算就完成了；

c) 要是某个被消元行变成了空行，就把它丢弃，不再参与后续的消去计算；如果其首项被当前批次覆盖，可是没有对应的消元子，就把它 “升格” 为消元子，在后续的消去计算中以消元子的身份，而不再以被消元行的身份参与；不断重复上述过程，直到所有批次都处理完，这时消元子和被消元行共同构成结果矩阵——或许存在不少空行。

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

```
void serial() {
    int begin = 0;
    int flag;
    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 不同算法和测试样本下的运行用时

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

3.4.2 实验结果总结

随着测试样本编号递增，问题规模亦逐步扩大。初期优化效果未显著体现，甚至呈现时间增加的状况，然而后续优化成效愈发显著，并且 8 路向量化的优化效果优于 4 路向量化。

4 链接

github 项目链接 GitHub