南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程实验报告

姓名：张天歌

学号 :2211123

专业：计算机科学与技术

2024 年 4 月 28 日

目录

Abstract

本次实验对普通高斯消去算法和特殊高斯消去算法进行了 SIMD 并行化研究，分别探索了两种算法在 neon、sse、avx、avx512 指令集下的性能，以及编译选项优化，对怎样实现高效并行进行了思考，并将研究结果记录下来，作为期末报告的一部分参考材料。

1 普通高斯消元

1.1 问题分析

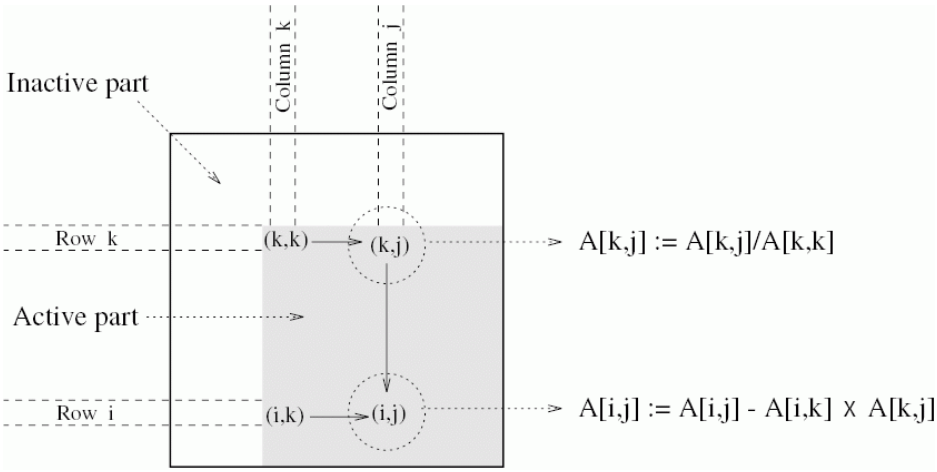

图 1.1: 高斯消去法示意图

高斯消去的计算模式如图 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。串行代码的算法如下面伪代码（算法 1）所示：

Algorithm 1 Gaussian Elimination 串行算法

```
1:  procedure GAUSSIAN_ELIMINATION(A,B)
2:      n ← size(A)                                          ▷ 消元过程
3:      for k from 1 to n do
4:          for i from k + 1 to n do
5:              factor ← A[i,k] / A[k,k]
6:              for j from 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: end procedure
```

伪代码第 5 行第一个内嵌循环中的 factor := A[k, j]/A[k, k] 以及伪代码第 6，7，8 行双层 for 循环中的 A[i, j] := A[i, j] − factor×A[k, j] 都可以进行向量化的循环化，因此可以通过 SIMD 扩展指令对这两步进行并行优化。

1.2 实验设计

基于 SIMD Intrinsics 函数对普通高斯消元进行向量化的伪代码（算法 2），可以逐句翻译为 Neon/SSE/AVX512 高斯消元函数，完成 ARM 和 X86 平台的基本实验，补齐测试样例生成和时间测量辅助工具，分别进行对齐和不对齐的 SIMD 访存操作。

Algorithm 2 SIMD Intrinsic 版本的普通高斯消元

Require: 系数矩阵 $A[n,n]$
Ensure: 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 $\leq n; j+=4$ **do**
    $va \leftarrow$ load4FloatFrom($\&A[k,j]$) ▷ 将四个单精度浮点数从内存加载到向量寄存器
    $va \leftarrow va/vt$ ▷ 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
  end for
  for j in remaining indices **do**
    $A[k,j] \leftarrow A[k,j]/A[k,k]$ ▷ 该行结尾处有几个元素还未计算
  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 $\leq n; j+=4$ **do**
      $vakj \leftarrow$ load4FloatFrom($\&A[k,j]$)
      $vaij \leftarrow$ load4FloatFrom($\&A[i,j]$)
      $vx \leftarrow vakj \times vaik$ ▷ 这里应该是向量对位相乘
      $vaij \leftarrow vaij - vx$
      store4FloatTo($\&A[i,j], vaij$)
    end for
    for j in remaining indices **do**
      $A[i,j] \leftarrow A[i,j] - A[k,j] \times A[i,k]$ ▷ 修正了这里的计算
    end for
    $A[i,k] \leftarrow 0$
  end for
end for

对于本问题，在初始化数据时，我们设计了一个 reset 函数，令主对角线上的元素 m1[i][i] = 1，令上三角区域的数据 m1[i][j] = rand()%1000 + 1, 下三角区域的数据 m1[i][j]+ = m1[k][j]。这种初始化可以重复生成样例、避免产生无穷数、检查程序正确性等。

在对串行算法进行优化的过程中，我们分别在 ARM 和 x86 架构上利用了 NEON、SSE、AVX 以及 AVX512 指令集，以评估不同硬件平台和指令集对算法并行化效果的影响。此外：

1. 考虑到 cache 层级和容量的限制，调整了问题规模 n 并观察了不同缓存大小对程序性能的影响，以分析缓存对性能的作用。

2. 还包括了对编译器优化级别影响的讨论，以及并行化策略对结果的具体影响。

3. 还考察了数据对齐与否对算法性能的差异。

详细的实验内容将在后续部分详细说明。

1.3 ARM 平台——NEON 指令集

1.3.1 实验内容

在本次实验中，我们主要关注了不同问题规模下，程序性能与缓存大小之间的关系。实验中使用的矩阵数组 m1[i][j] 采用 float 类型，每个元素占用 4 字节。给定 float 类型变量的总数为 N，以及第 i 级缓存的大小为 Si，我们有 $N\times4 = S_i, i = 1, 2, 3$，其中 i 可以是 1, 2, 或 3，代表不同的缓存级别。

选项	信息
内核版本	Linux master 4.14.0-115.el7a.0.1.aarch64
发行版版本	OpenEuler(CentOS)
编译器	HUAWEI 毕升编译器 (clang 12.0.0)
CPU 型号	鲲鹏 920 服务器版
性能分析工具	Perf
CPU 核心数	96 核 96 线程
CPU 主频	2.6GHz
L1 一级缓存	96 64KB 数据缓存 96 64KB 指令缓存
L2 二级缓存	96 512KB
L3 三级缓存	48MB
内存	20GB

表 1: 鲲鹏服务器的 CPU 相关参数

针对本实验，当问题规模 n 足够大时，可以近似认为 $n^2 \approx N$ 。参照表1，通过将 S1、S2、S3 分别代入上述公式，我们估算出填满各级缓存对应的 n 大约在 120-130、350-400 和 3500-3600 之间。为了观察不同缓存大小对性能的影响，我们选择了 n 在 100 到 1000 之间每隔 100 个单位的值，以及 2000、3000、4000 这三个点，共计 13 个不同的问题规模。这样的取值不仅保证了 n 为 4 的倍数，而且由于我们将数组 m1[i][i] 设置为全局变量，实现了二维数组在每一维度上的 16 字节地址对齐，简化了地址对齐的处理。实验步骤如下：

1. 在本地编辑器中完成了普通高斯消元的串行代码编写。
2. 参考实验手册和 ARM Developer 网站的指令集指南，编写了适用于 NEON 指令集的并行不对齐代码。
3. 分别实现了仅除法部分并行化和仅减法部分并行化的代码。
4. 在不对齐基础上，完成了数据对齐版本的代码。

实验通过比较串行算法与并行算法的耗时比值，来评估优化效果。实验结束后，对结果进行了综合分析，以揭示性能优化的内在原因。

1.3.2 编程实现

为了分析向量化对循环内部的除法归一过程和消元过程所造成的优化效果的不同，设计在不同步骤分别采用向量指令的策略。代码如下：

ARM 指令集并行化

```
for(int k=0;k<n;k++){
//除法部分向量化
```

```
        float32x4_t vt=vld1q_dup_f32(m1[k]+k);
        int j;
        for(j=k;j+4<=n;j+=4){
            float32x4_t va =vld1q_f32(m1[k]+j);
            va=vdivq_f32(va,vt);
            vst1q_f32(m1[k]+j,va);
        }
        for(;j<n;j++)
            m1[k][j]=m1[k][j]/m1[k][k];
        m1[k][k]=1.0;
        for(int i=k+1;i<n;i++){
    //消元部分向量化
            float32x4_t vaik=vld1q_dup_f32(m1[i]+k);
            int j;
            for(j=k+1;j+4<=n;j+=4){
                float32x4_t vakj=vld1q_f32(m1[k]+j);
                float32x4_t vaij=vld1q_f32(m1[i]+j);
                float32x4_t vx= vmulq_f32(vaik,vakj);
                vaij = vsubq_f32(vaij,vx);
                vst1q_f32(m1[i]+j,vaij);
            }
            for(;j<n;j++)
                m1[i][j]=m1[i][j]-m1[i][k] m1[k][j];
            m1[i][k]=0;
        }
    }
```

1.3.3 实验结果与分析

(1) 对于整体并行化非对齐算法:

在鲲鹏服务器上无优化的情况下对程序进行编译并运行，以串行算法的耗时与并行算法的耗时之比作为优化力度评估优化算法的影响。对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。结果如下表2所示:

在分析串行与并行算法的耗时比数据时，我们观察到耗时比在大部分情况下维持在 1.5—1.6 之间，但在问题规模 n=700 时出现显著下降至大约 1.2。这一现象并未与任何特定缓存层级的填满直接相关，表明 L1 或 L2 缓存的填满并非是影响性能优化的主要因素。通过修改初始数据并观察到耗时比保持不变，我们排除了特殊数据对性能影响的可能性。进一步分析发现，在 n=500 至 n=600 的过程中，串行算法耗时增加了 0.24 秒，而从 n=600 增至 n=700 时，耗时仅增加了 0.12 秒。这与通常随 n 增大耗时差额也增大的规律不符，表明在 n=700 时，串行算法的效率异常提高，而非并行算法效率降低，导致耗时比下降。

使用 perf 工具进行深入分析后，我们发现在 n=700 时，串行算法的缓存未命中（cache miss）率有所下降，这是其性能提升的原因。此外，在 n=3000 时，耗时比也有所下降，通过在 n=3500 处增加额外测试，我们确认耗时比维持在大约 1.18，这与填满 L3 缓存所需的数据规模相匹配，表明此时性能下降受到了 L3 缓存的影响。

综上所述，耗时比的变化揭示了算法性能在特定问题规模下的异常表现（如图1.2），并通过工具分

数据规模	串行用时/s	并行用时/s	加速比
100	0.00247	0.00161	1.513
200	0.0194	0.0127	1.532
300	0.0669	0.0425	1.574
400	0.1547	0.1005	1.540
500	0.3069	0.2001	1.534
600	0.5499	0.3494	1.573
700	0.6676	0.5533	1.207
800	1.2539	0.8266	1.517
900	1.7980	1.1682	1.539
1000	2.4794	1.6187	1.532
2000	20.028	12.949	1.547
3000	53.197	46.430	1.146
4000	176.631	121.125	1.458

表 2: NEON 非对齐性能测试结果

析找到了性能提升和下降的具体原因。

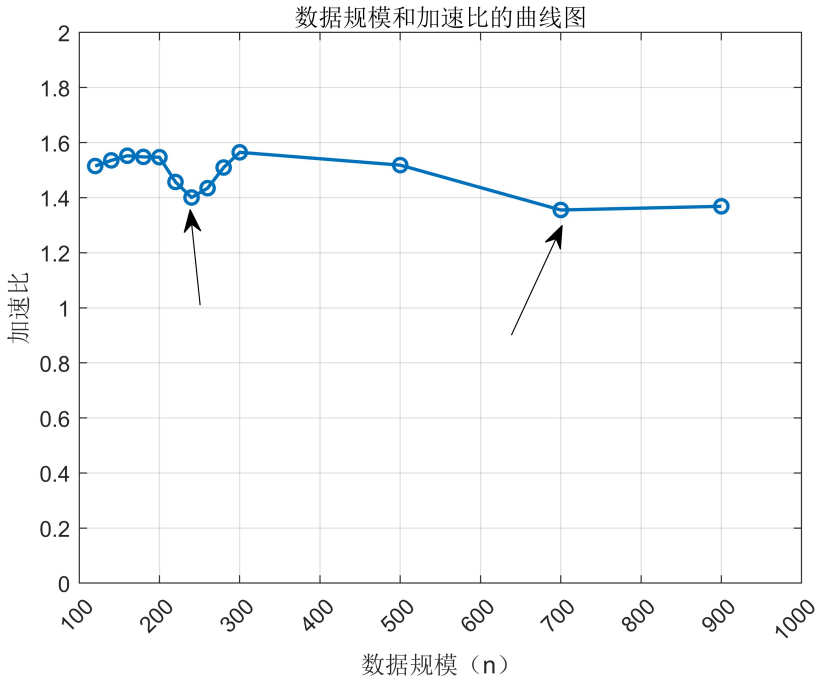

图 1.2: NEON 未对齐并行化

（2）对于整体并行化对齐算法：

对齐实验原理：需要对上三角矩阵 m1 进行对齐处理，对每一行的前几个元素进行单独处理，直到找到该行的“主元”（主元是行中第一个非零元素，它在上三角矩阵中尤为重要）。列数倍数对齐指的是继续处理矩阵直到当前处理的元素的列数是 4 的倍数，如果处理到的列数不是 4 的倍数，可能需要对行中的元素进行交换或移动，以确保列数符合对齐要求。原理如下图1.3表示：

对代码进行修改后，编译和运行的操作同上，结果如下表3所示：

同理制作曲线图，画出加速比和规模的相应关系，如图。两条曲线走势大致相同，但是发现在相同规模下，对齐后的加速比相比于未对齐的平均多出 0.4 秒，观察数据发现串行耗时与未对齐时几乎

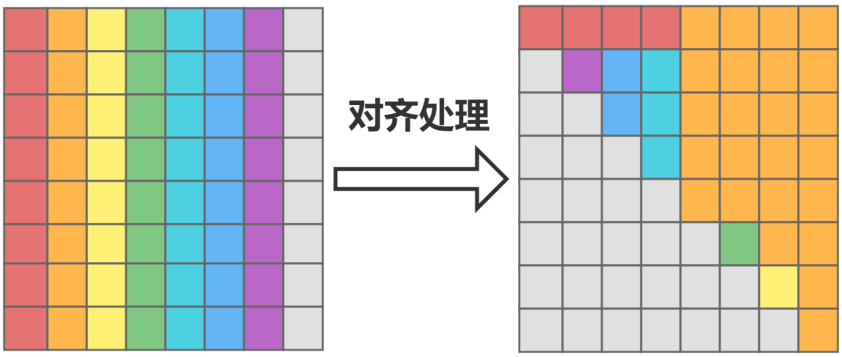

图 1.3: 对代码进行对齐处理

数据规模	串行用时/s	并行用时/s	加速比
100	0.00244	0.00129	1.898
200	0.0194	0.0099	1.947
300	0.0654	0.0333	1.967
400	0.157	0.0797	1.973
500	0.314	0.159	1.969
600	0.547	0.276	1.979
700	0.869	0.441	1.968
800	1.303	0.660	1.972
900	1.852	0.938	1.975
1000	2.530	1.280	1.976
2000	19.839	9.957	1.992
3000	70.74	36.255	1.951
4000	202.753	109.174	1.857

表 3: NEON 对齐性能测试结果

相同，而并行耗时得到了缩短，因此对齐的作用在于更好的优化了 NEON 指令集的并行化，而未对其他因素有明显影响。

（3）对于部分并行化的算法：

在对齐条件下，对仅除法部分和仅减法部分分别进行并行化：矩阵同一行进行除法部分，总运算次数约为 $\frac{n^2}{2}$，即时间复杂度为 $O(n^2)$；多行间减法的第二部分，总运算次数达到 $O(n^3)$ 的级别。

对除法部分进行并行化后，发现串行算法和并行算法的耗时几乎相同（如图5(a)），不同问题规模下的加速比都维持在 1 左右；再对减法消去部分进行并行化（如图5(b)），发现并行算法的耗时以及串并行算法的加速比与整体并行化时几乎一致为 1.9 左右。因此可以得出结论：在 ARM 平台的 NEON 指令集下，是否对时间复杂度更小的第一部分进行并行化几乎不会影响结果，对时间复杂度较大的一部分并行化才能起到显著得并行加速的效果。

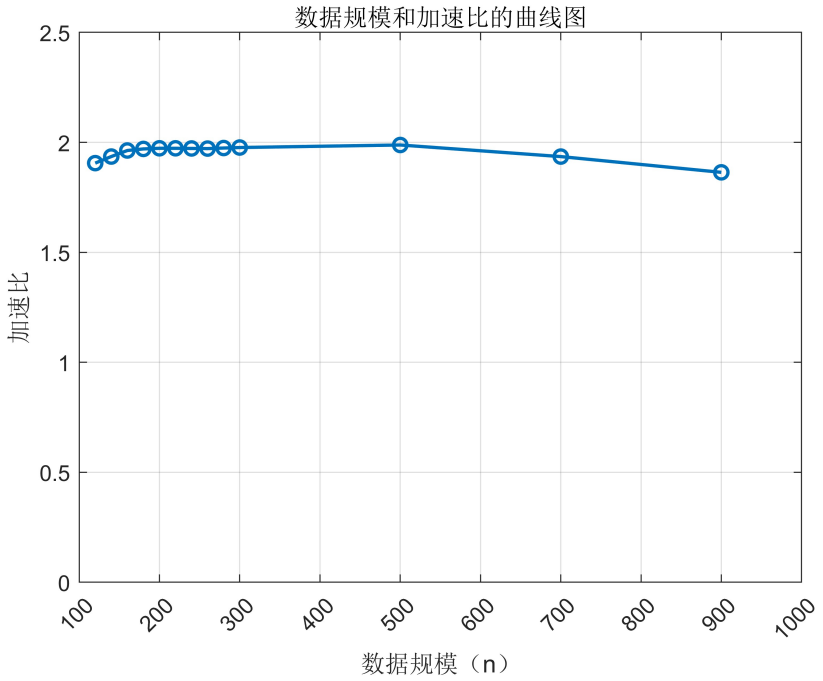

图 1.4: NEON 对齐后并行化

```
800
total: 1.26615 num: 1 serial: 1.26615
total: 1.26966 num: 1 parallel: 1.26966
0.997234
900
total: 1.80002 num: 1 serial: 1.80002
total: 1.80806 num: 1 parallel: 1.80806
0.995555
1000
total: 2.45462 num: 1 serial: 2.45462
total: 2.46457 num: 1 parallel: 2.46457
0.995965
```

(a) 仅除法的并行化

```
N为：800
total: 1.30023 num: 1 serial: 1.30023
total: 1.31572 num: 2 parallel: 0.657859
1.97645
N为：900
total: 1.84024 num: 1 serial: 1.84024
total: 1.87617 num: 2 parallel: 0.938086
1.9617
N为：1000
total: 2.53566 num: 1 serial: 2.53566
total: 1.27752 num: 1 parallel: 1.27752
1.98483
```

(b) 仅减法的并行化

图 1.5: 对于部分并行化的实验结果

1.4 X86 平台——SSE、AVX、AVX512 指令集

1.4.1 实验内容

在对比 ARM 平台的鲲鹏服务器和 x86 的缓存架构时（如表4），注意到后者的 L1 缓存较小，缺少 L3 缓存，但拥有更大的 L2 缓存。根据缓存大小和问题规模的关系，我们估算出填满 L1 缓存的 n 大约在 90-100 范围内，而填满 L2 缓存的 n 则在 1000-1100 之间。

为了深入分析问题规模与缓存大小对程序性能的影响，我们选择了 n 为 64、128、256、512、1024、2048，共 6 个不同的问题规模，确保 n 为 16 的倍数。二维数组的每个维度都进行了相应的字节对齐处理，简化了地址对齐的复杂性，这样取数可以使 n 都为 16 的倍数，配合了对地址对齐要求最高的 avx512 指令。

实验步骤如下：

1. 在 VS Code 中编写了针对 SSE、AVX 指令集的代码，利用 Intel Intrinsics Guide 网站的指导，通过微小的改动实现不同指令集间的过渡。

2. 对代码的不同部分进行了并行化处理，包括除法、减法，以及考虑了数据对齐与否的情况。

选项	信息
CPU 型号	Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
内存	192GB
CPU 主频	基准频率 3.40GHz 最高频 3.90GHz
CPU 核心数	24 核 48 线程
L1 一级缓存	384KB 指令 384KB 数据
L2 二级缓存	12MB
L3 三级缓存	38.5MB
内存	20GB

表 4: Intel Cloud 服务器上的 CPU 相关参数

3. 最后对实验结果进行了详尽的总结和分析，以评估并行化及对齐优化对性能的具体影响，大致实现流程与 ARM 平台下类似。

实验结果的分析旨在揭示不同缓存层次和问题规模对程序性能的影响，以及不同指令集并行化策略的效果。

1.4.2 编程实现

在 x86 平台上，有 intel 提供的原生 SIMD 指令支持。比如相继推出的 MME（支持 64 位寄存器），SSE（支持 128 位寄存器），AVX（支持 256 位的寄存器），AVX-512（支持 512 位寄存器）指令架构。编译器可以将 intrinisc 函数编译为相应 SIMD 的汇编代码，在寄存器级别实现向量运算指令，可以成倍地提高运算速度。本机测试时将使用本地 Code::Blocks 开发工具。

X86 指令集并行化

```
    for(int k=0;k<n;k++){
        float temp[4]={m1[k][k],m1[k][k],m1[k][k],m1[k][k]};
        __m128 vt= _mm_loadu_ps(temp);
        int j;
        for(j=k;j+4<=n;j+=4){
            __m128 va =_mm_loadu_ps(m1[k]+j);
            va=_mm_div_ps(va,vt);
            _mm_storeu_ps(m1[k]+j,va);
        }
        for(;j<n;j++)
            m1[k][j]=m1[k][j]/m1[k][k];
        m1[k][k]=1.0;
        for(int i=k+1;i<n;i++){
            float temp2[4]={m1[i][k],m1[i][k],m1[i][k],m1[i][k]};
            __m128 vaik=_mm_loadu_ps(temp2);
            int j;
            for(j=k+1;j+4<=n;j+=4){
                __m128 vakj=_mm_loadu_ps(m1[k]+j);
                __m128 vaij=_mm_loadu_ps(m1[i]+j);
                __m128 vx= _mm_mul_ps(vaik,vakj);
                vaij = _mm_sub_ps(vaij,vx);
                _mm_storeu_ps(m1[i]+j,vaij);
```

```
            }
            for(;j<n;j++)
                m1[i][j]=m1[i][j]-m1[i][k] m1[k][j];
            m1[i][k]=0;
        }
    }
```

1.4.3 实验结果与分析

（1）对于整体并行化非对齐算法：

在 Intel Devcloud 上无优化的情况下对程序进行编译并运行，以串行算法的耗时与并行算法的耗时之比作为优化力度评估优化算法的影响。对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。结果如下表所示:

数据规模	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s	串行/SSE	串行/AVX	串行/AVX512
64	0.00040	0.00019	0.000151	0.000156	2.099	2.667	2.579
128	0.00327	0.0014	0.000983	0.000799	2.279	3.327	4.094
256	0.026	0.011	0.00713	0.00476	2.354	3.643	5.459
512	0.206	0.0854	0.0541	0.032	2.416	3.815	6.435
1024	1.676	0.705	0.429	0.237	2.378	3.901	7.064
2048	13.373	5.563	3.364	1.926	2.404	3.976	6.942

表 5: x86 非对齐性能测试结果

分析图（如图1.6）与表中的数据可以发现，并行化的效果为 avx512>avx>sse。其中，sse 算法的效能在问题规模 n 较小时随规模增大略有提高，之后趋于稳定，与串行算法的耗时比约在 2.3、2.4 左右；avx 算法的效能随问题规模的增大要强于 sse 算法，同样是先增大后稳定，最后耗时比可达到 4；avx512 算法受问题规模影响最大，在 n 较小时性能的变化最为显著，在 n 较大时可达到的稳定耗时比也最高，7 左右。

图中两条竖线分别代表 L1 cache、L2 cache 对应的问题规模，可以发现在 L1 cache 填满前，曲线大幅上扬，优化效果明显提升；在填满 L1 cache、未填满 L2 cache 时，曲线缓慢攀升，并行优化结果略有上升；在 L2 cache 填满后，曲线变得平缓，优化效果达到平衡。可以得出结论：sse、avx、avx512 三种指令集下的并行化受问题规模的影响方式相似，影响程度不同，并且问题规模与各级 cache 大小的对应关系是影响并行优化效果的主要因素。

（2）对于整体并行化对齐算法：

由于 X86 平台下的指令集在对齐的情况下使用的指令与非对齐时略有不同，因此还要进行指令的修改。对代码进行修改后，编译和运行的操作同上，结果如下表所示：

制作曲线图（如图1.7）以展示不同指令集与串行算法耗时比，观察出：

1. 对齐效果：数据对齐后，性能提升在各个问题规模下都有所增加，特别是在大规模数据下，优化效果更为显著。2. 性能提升：SSE 指令集的性能提升约为 0.9 倍，AVX 指令集的性能提升约为 1 倍，AVX512 指令集的性能提升约为 1 倍。3. 串行耗时：串行算法的耗时与未对齐时相比几乎没有变化。4. 并行耗时：并行算法的耗时得到了缩短，表明数据对齐主要优化了算法的并行执行部分。5. 其他因素：数据对齐对于 cache 规模等其他因素的影响与未对齐时相同，没有产生额外影响。

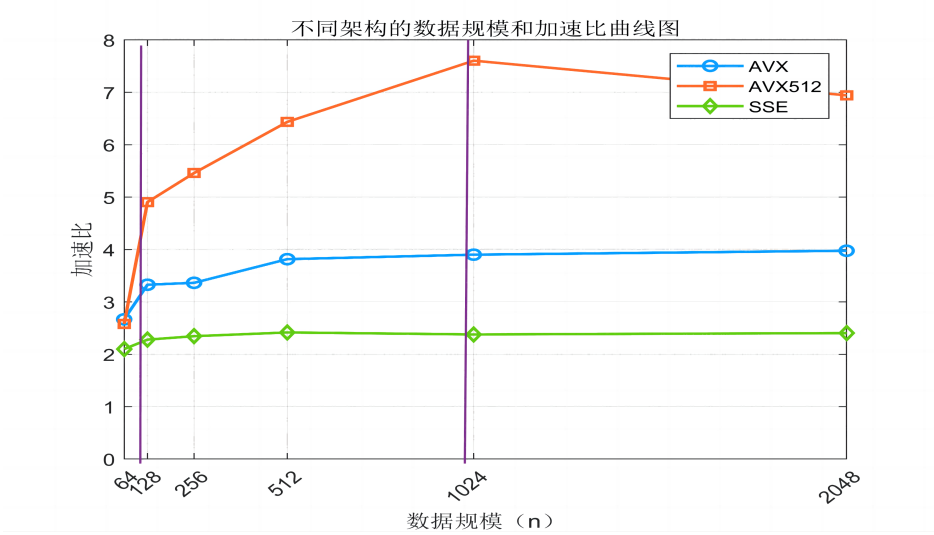

图 1.6: x86 非对齐并行化

数据规模	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s	串行/SSE	串行/AVX	串行/AVX512
64	0.000401	0.000151	0.000144	0.000157	2.651	2.78	2.55
128	0.00329	0.0011	0.000887	0.000810	2.999	3.747	4.055
256	0.0258	0.00792	0.006	0.00446	3.257	4.3	5.687
512	0.207	0.0625	0.0437	0.0288	3.312	4.73	7.086
1024	1.672	0.501	0.338	0.203	3.337	4.95	8.06
2048	13.213	3.878	2.562	1.744	3.406	4.97	8.02

表 6: x86 对齐性能测试结果

综上所述，数据对齐主要通过优化并行算法的执行来提升性能，而对串行算法的耗时和其他因素如 cache 规模等没有显著影响。

（3）对于部分并行化：

对时间复杂度为 $O(n^2)$ 的第一部分进行并行化后，发现并行算法的耗时仅比串行算法略短，不同问题规模下的耗时比都维持在 1.2-1.3; 再对时间复杂度为 $O(n^3)$ 的第二部分进行并行化，并行算法的耗时比串行算法显著缩短（虽比不上整体并行化时的优化效果），如 n=128 时串行算法与 sse、avx、avx512 的耗时比分别为 2.17、3.206、3.969。

因此可以得出结论：X86 平台下的各大指令集，对于串行算法的并行化效果同时受到除法部分和减法消去部分两者的影响，第一部分复杂度低，优化效果较弱；第二部分复杂度高，优化效果较强，同时对两部分进行并行化才能取得最佳优化结果。

1.5 普通高斯消元实验总结与优化尝试

1.5.1 ARM 与 X86 平台对比

综合上述实验数据，普通高斯消元法在 ARM 平台下和 X86 平台下存在诸多差异，大致有如下几点：

- 指令集差异：X86 平台下的各指令集的并行化效果优于 ARM 平台的 NEON 指令集
- 缓存影响：X86 平台下的算法受各级 cache 的影响较为明显，而 ARM 平台下仅受 L3 cache 的

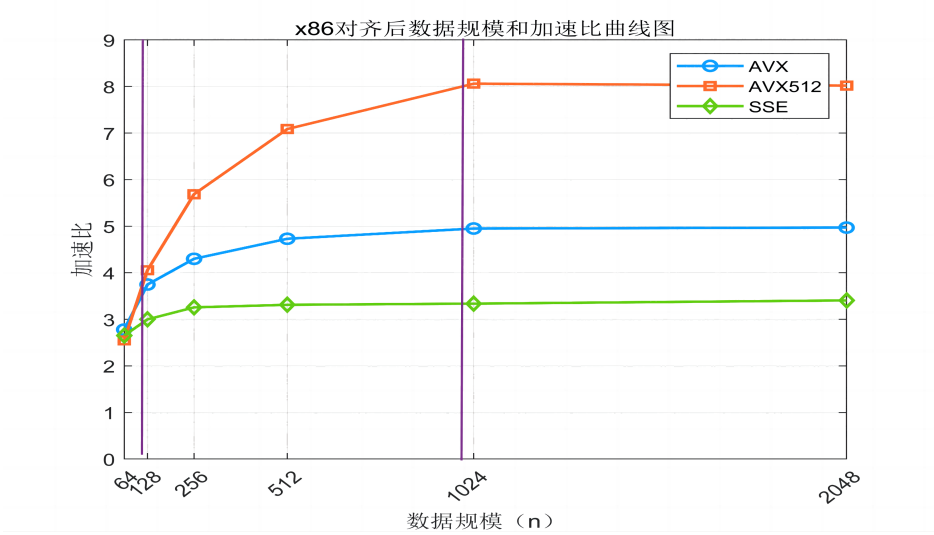

图 1.7: x86 对齐并行化

影响。

- 对齐 VS 不对齐：对齐后 X86 和 ARM 平台下的并行效果都得到提升，从绝对数值上看 X86 平台下提升更大，从相对比例而言 ARM 平台下提高更多
- 部分并行化：ARM 平台下的并行化提升几乎完全取决于消去的第二部分的并行化，而 X86 平台下的并行化性能由两个部分共同决定，第一部分影响小，第二部分影响大。

1.5.2 编译选项的优化

分别在 ARM 平台和 neon 平台下依次尝试分别采用-O1、-O2、-O3 参数来代替原有的-O0 参数进行优化。在 ARM 平台下，串行算法的运行速度是 O0<O1<O2<O3，并行算法则是 O0<O1<O2=O3，这就导致了串并行耗时比先增大后减小，O3 优化后甚至达到 1:1。这是由于低级的编译参数对串并行算法都有优化，且并行优化更多，而高级编译参数仅对串行算法起优化作用。在 X86 平台下，串行算法的运行速度是 O0<O1=O2<O3，并行算法则是 O0<O1<O2=O3，虽然与 ARM 平台下略有不同，但耗时比同样先增大后减小。总结可以发现在两个平台下优化编译选项都能提升运行速度，且低级优化选项对于并行算法提升更多，高级优化选项对串行算法提升更多。具体耗时比变化可见于下图1.8，其中 arm 平台下为 n=500，X86 平台下 n=128，其他规模下的变化与此类似：

2 基于 Gröbner 基计算的特殊高斯消元

2.1 问题分析

特殊高斯消元算法在有限域 GF(2) 上进行，主要涉及两个操作：消元子和被消元行的操作。消元子是用于消去其他行的行，而不会被其他行消去。被消元行则是在消元过程中被消去的行，但它们有可能因为包含特定的对角线元素而变成消元子。该特殊高斯消元的算法流程如下：

1. 将文件中的消元子和被消元行读入内存

2. 对每个被消元行，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）或首项在范围内但无对应消元子或该行，若为情况 2 则该行计算完成

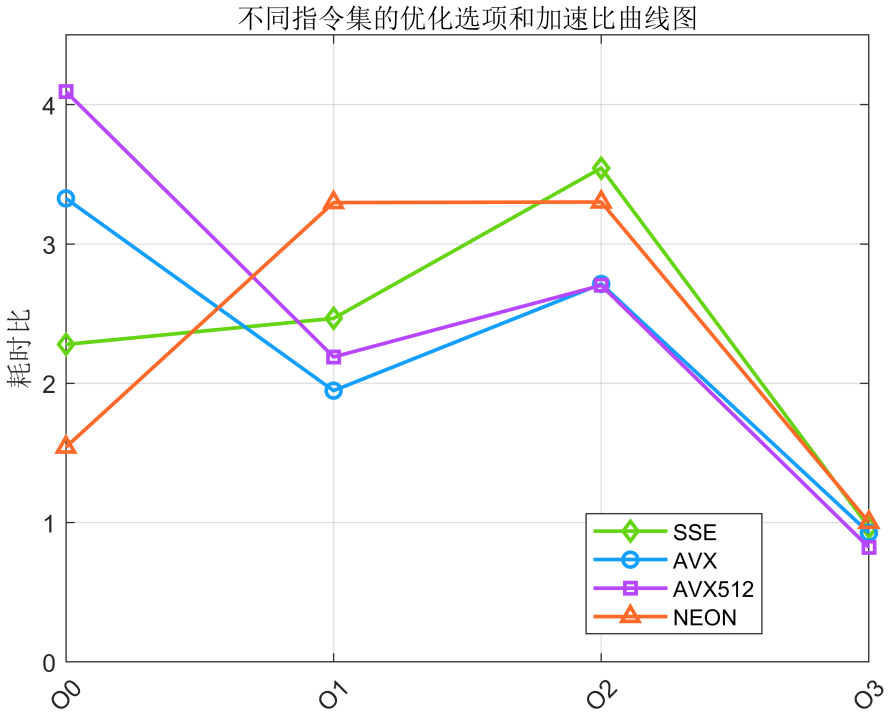

图 1.8: 不同指令集的优化选项和加速比曲线图

3. 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

4. 重复第二步和第三步的过程，直至所有被消元行都处理完毕，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。其串行算法的代码如下所示，其中 R[i] 代表首项为 i 的消元子，E[i] 代表第 i 个被消元行，lp(E[i]) 代表被消元行第 i 行的首项。

其串行算法的代码如下所示，其中 R[i] 代表首项为 i 的消元子，E[i] 代表第 i 个被消元行，lp(E[i]) 代表被消元行第 i 行的首项。

Algorithm 3 串行特殊高斯消元算法

```
for i from 0 to m − 1 do
    while E[i] ≠ 0 do
        x ← lp(E[i])
        if R[x] ≠ NULL then
            E[i] ← E[i] − R[x]
        else
            R[x] ← E[i]
        end if
        break
    end while
end for
return E
```

2.2 实验设计

对于本问题，我们设计了一个位向量类用于存储每行消元子/被消元行相关信息。类中存在两个成员变量，一个 int 型数据用来记录当前行的首项所在的列，一个 int 型数组采用位向量方式存储该行消元子或被消元行中的元素。除此之外，重要的功能还有类内实现的异或功能、类外实现的数据读取功能与消元功能等等。与普通高斯消元的做法类似，本实验对问题规模 n 进行了多次改变，观察了问题规模与 cache 大小不同关系时的程序性能变化，分析 cache 对于性能的影响，揭示性能表现的内在原因。另外，实验中还对于编译器不同优化力度对于性能的影响、对齐与不对齐算法的差异进行简要讨论。

2.3 X86 平台——SSE、AVX、AVX512 指令集

2.3.1 算法设计

本实验的 X86 平台实验同样在 Intel Devcloud 服务器上进行，该服务器的 CPU 参数已在普通高斯消元部分给出。

同样采用上一节中对应 cache 的问题规模大小计算公式，可以得到此时填满 L1 cache 对应的 n 约在 500 左右，填满 L2 cache 对应的 n 约在 5700-5800 之间。沿用之前样例中的 7 组数据，其中有 2 组数据小于 L1 cache 对应的规模，4 组数据在 L1 和 L2 cache 之间，1 组数据大于 L2 cache。继续采用了 aligned-alloc() 函数来保证每个实例内的数组都可完成 64(32、16) 字节地址对齐，以减少后续处理地址对齐的手续。

基于以上数据,开始进行代码的编写。代码的编写步骤同样与上一节相似,并且 SSE、AVX、AVX512 指令集的代码重合度极高，根据 Intel IntrinsicsGuide 网站的指导，只需要微小改动便可由一个指令集过渡到另一个指令集。我们同样对是否对齐的情况进行了实验，并对结果进行了总结分析。

2.3.2 实验结果与分析

(1) 对于非对齐算法：

在 Intel Devcloud 上无优化的情况下对程序进行编译并运行，每个问题规模下的实验重复运行 5 次来取平均值。结果如下表所示：

数据规模	读取数据用时/s	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s
130	0.000252	0.0000615	0.00006	0.0000606	0.0000627
254	0.000812	0.000703	0.000699	0.000671	0.000715
562	0.00143	0.00115	0.00102	0.00109	0.00119
1011	0.00715	0.0401	0.0386	0.0369	0.0427
2362	0.0197	0.321	0.301	0.317	0.353
3799	0.106	7.801	7.681	7.435	8.092
8399	0.671	108.597	105.672	103.207	109.26

表 7: x86 特殊高斯消元的非对齐并行化

在小数据规模下，数据读取的耗时在总耗时中占有显著比例，有时甚至超过算法执行的耗时，随着数据规模的增加，数据读取耗时的增长速度相对缓慢，对总耗时的贡献逐渐减小，可以被忽略。SSE、AVX 和 AVX512 架构的耗时曲线变化趋势相似，表明这些架构对算法性能的影响具有一定的共性。SSE 和 AVX 架构对提升算法性能有轻微的正面影响，但效果有限；相比之下，AVX512 架构在本实

验的并行化尝试中并未展现出优势，其耗时反而超过了串行算法。曲线变化的规模与不同级别的 cache 大小没有直接对应关系，这表明在 X86 平台上，cache 大小对算法性能的影响可能并不显著。

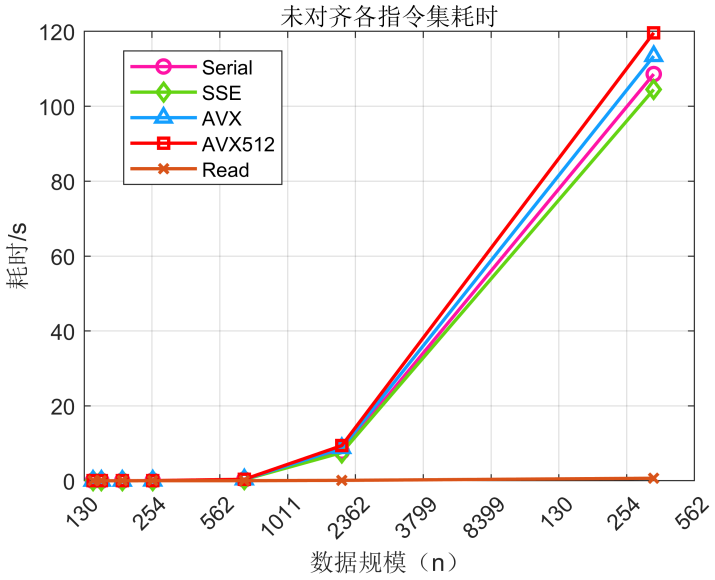

图 2.9: 特殊高斯消元非对齐并行化

（2）对于对齐算法：

大致处理思路与前一实验相同，即先进行对齐运算，最后对每行多出的几个元素单独处理。与非对齐时最大的差别在于采用了 aligned-alloc() 函数来对齐地址。由于 X86 平台下的指令集在对齐的情况下使用的指令与非对齐时略有不同，因此还要进行指令的修改。对代码进行修改后，编译和运行的操作同上，结果如下表所示：

数据规模	读取数据用时/s	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s
130	0.000252	0.0000615	0.00008	0.0000677	0.0000627
254	0.000812	0.000703	0.000725	0.00071	0.000745
562	0.00143	0.00115	0.00112	0.00108	0.00116
1011	0.00715	0.0401	0.0369	0.0417	0.0447
2362	0.0197	0.321	0.307	0.349	0.393
3799	0.106	7.801	7.525	8.716	9.385
8399	0.671	108.597	104.473	113.407	119.62

表 8: x86 特殊高斯消元的对齐并行化

下面曲线图将展示不同指令集架构相对于串行算法的耗时比，揭示并行化对性能的影响。尽管进行了地址对齐，但曲线的走势与未对齐时相似，暗示对齐操作并未显著改变算法性能的基本趋势。SSE 指令集的并行算法在对齐后显示出略微的性能提升，耗时有所减少。相比之下，AVX 和 AVX512 指令集在对齐操作后耗时增加，甚至超过了串行算法的耗时，表明这些指令集在当前实验中并未提供性能优势。额外的时间开销可能是由于 API 函数调用进行地址对齐时产生的，这抵消了并行化带来的潜在好处。

综合分析表明，对于本程序，AVX 和 AVX512 指令集并未实现有效的性能优化，而 SSE 指令集

在并行化方面表现最佳，尽管其优化效果有限。对于 cache 规模等因素，结论与未对齐时相同，即它们对算法性能的影响不明显。

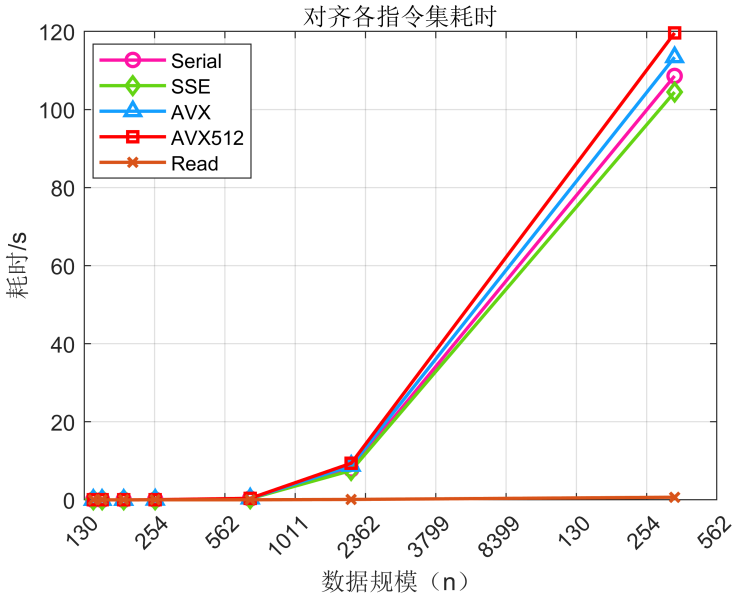

图 2.10: 特殊高斯消元对齐并行化

2.4 优化编译选项

在 X86 平台上，通过调整编译优化参数（-O1、-O2、-O3），可以显著提升程序的运行速度，以下是实验总结：

1. 在 X86 平台上，串行算法的优化效率排序为 -O0 < -O2 = -O3 < -O1，而并行算法的排序为 -O0 < -O3 < -O2 < -O1。

2. 优化编译选项在 x86 上能显著提升程序运行速度。例如，当问题规模 col=3799 时，在 X86 平台使用-O1 参数时，速度提升约 3 倍。

3. 不同指令集之间优化速率略有不同，SSE,AVX 优化效果较差，平均提速为 2 倍左右；AVX512 优化效果较好，最高提速可达到 4 左右。

3 总结与反思

本次实验发现普通高斯消元算法非常适合 X86 平台、avx512 指令集下的地址对齐并行化，对于 neon、sse、avx 指令集下的并行化也能获得一定的性能提升。而特殊高斯算法仅在 neon、sse 指令集下进行并行化时可以获得细微的性能提升。这部分成果将与此后除 SIMD 外的其他并行化实验进行比较分析，作为期末报告的参考资料。

另外，对于特殊高斯算法还有许多优化的空间，例如分批次读取处理、除异或操作外其他部分的并行化、消元顺序变更、计算和访存异步等等。这些探究同样将在日后的实验中进行操作与完善。

代码详见：https://github.com/TIANGE2211123/parrallelism/tree/main/SIMD