南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 加速的高斯消去算法

姓名：熊宇轩

学号：2010056

专业：计算机科学与技术

2023 年 6 月 2 日

目录

1 实验目标

1. 编写代码实现普通高斯消去算法以及消元子模式的高斯消去算法，并通过 intrinsic 方式利用 X86 和 ARM 平台上 NEON、SSE、AVX、AVX512 等指令集实现 SIMD 加速。
2. 编写测试脚本，通过运行计时的方式，测试不同规模下不同加速算法的加速比。
3. 利用 perf 等工具对实验结果进行分析，找出性能差异的原因。

2 核心代码

2.1 普通高斯消去算法

2.1.1 串行算法

串行算法的实现比较简单，代码如下：

```
#define N 1024
#define ele_t float

void LU(ele_t mat[N][N], int n)
{
    memcpy(new_mat, mat, sizeof(ele_t) * N * N); // 复制矩阵

    for (int i = 0; i < n; i++) // 遍历行
        for (int j = i + 1; j < n; j++)
        { //遍历列
            if (new_mat[i][i] == 0) // 当前元素已经为 0，不需要消去
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i]; // 一次性计算除数
            for (int k = i; k < n; k++)
                new_mat[j][k] -= new_mat[i][k] * div;
        }
    // 省略：用于输出矩阵验证正确性的代码
}
```

2.1.2 NEON/SSE 加速

NEON 的加速实现也较为简单。不过，在实现 NEON 加速的过程中，作者发现了一个英特尔提供的，可以将 NEON 代码迁移至 SSE 加速的头文件。[2] 虽然对于此次作业来说，这个工具减少的代码量微乎其微，但这极大方便了程序的调试。

之所以能通过一个简单的头文件在 X86 平台上运行 ARM NEON 的代码，主要是因为 SSE 和 NEON 指令集寄存器位数都是 128，且功能十分相似。NEON 和 SSE 相较于 AVX 和 AVX512 等指令集，使用 intrinsic 编程的方式也十分相似，虽然寄存器位数不同，但也只需要修改调用的函数和每次 load/store 的数据个数即可。

此外，在研究该头文件实现时，作者发现其中的融合乘加函数其实是将一条融合乘加指令拆成两条指令实现的。然而，SSE 指令集原生是支持融合乘加的，我们可以借此机会，研究融合乘加和非融合乘加 SIMD 对加速的影响。

关于对齐的问题，由于串行算法在实现矩阵的第 j 行减去第 i 行时，省略了第 i 列前的元素（在第 i 行 i 列前的元素已经全是 0 了），因此根据串行算法修改的来的 SIMD 算法“自然地”是不对齐的，通过将省略元素从 i 列前改为 i / 4 * 4 前，我们可以实现对齐的算法。

```
#ifdef __ARM_NEON
#include <arm_neon.h>
#endif

#ifdef __amd64__
#include <immintrin.h>
#include "NEON_2_SSE.h" // Intel 提供的用于将 NEON 代码迁移至 SSE 的头文件
#endif

#define ele_t float
ele_t new_mat[N][N] __attribute__((aligned(64))); // 对齐
ele_t mat[N][N];

void LU_simd(ele_t mat[N][N], int n)
{
    memcpy(new_mat, mat, sizeof(ele_t) * N * N); // 复制矩阵

    for (int i = 0; i < n; i++) // 遍历行
        for (int j = i + 1; j < n; j++)
        {                           //遍历列
            if (new_mat[i][i] == 0) // 当前元素已经为 0，不需要消去
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            float32x4_t mat_j, mat_i, div4 = vmovq_n_f32(div);
#ifdef ALIGN // 是否对齐
            for (int k = i / 4 * 4; k < n; k += 4)
#else
            for (int k = i; k < n; k += 4)
#endif
            {
                mat_j = vld1q_f32(new_mat[j] + k);
                mat_i = vld1q_f32(new_mat[i] + k);
#ifdef USE_FMA // 使用融合乘加
                vst1q_f32(new_mat[j] + k, _mm_fnmadd_ps(mat_i, div4, mat_j));
#else
```

```
                vst1q_f32(new_mat[j] + k, vmlsq_f32(mat_j, div4, mat_i));
#endif
            }
        }
}
```

2.1.3 AVX 加速

AVX 加速和 NEON/SSE 加速的编程极为相似，仅仅是在函数命名和参数上有微小区别。不过需要注意的是 AVX 的对齐和不对齐 load/store 使用的是不同的函数。

AVX512 的实现与 AVX 高度相似，这里不再赘述。

```
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
        {
            if (new_mat[i][i] == 0)
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            __m256 mat_i, mat_j, div8 = _mm256_set1_ps(div);
#ifdef ALIGN
            for (int k = i / 8 * 8; k < n; k += 8)
#else
            for (int k = i; k < n; k += 8)
#endif
            {
#ifdef ALIGN
                mat_j = _mm256_load_ps(new_mat[j] + k);
                mat_i = _mm256_load_ps(new_mat[i] + k);
                _mm256_store_ps(new_mat[j] + k, _mm256_fnmadd_ps(mat_i, div8, mat_j));
#else
                mat_j = _mm256_loadu_ps(new_mat[j] + k);
                mat_i = _mm256_loadu_ps(new_mat[i] + k);
                _mm256_storeu_ps(new_mat[j] + k, _mm256_fnmadd_ps(mat_i, div8, mat_j));
#endif
            }
        }
```

2.2 消元子模式的高斯消去算法

2.2.1 数据结构相关代码

在消元子模式的高斯消去算法中，消元子和被消元行矩阵采用位图方式存储，由于实验中使用的数据集的矩阵列数大多不是 4、8、16 的整倍数，这给我们提供了实验对齐和不对齐性能差距的机会。

因此作者设置了两行可以通过宏定义激活的代码，它将增加矩阵的行数到 16 的整倍数。

此外，由于数据集采用的是稀疏矩阵，而程序内部运算是位图格式，因此还需要设计读入和转换的代码。需要注意的是，为了方便消元子的查找以及将被消元子升格为消元子的操作，消元子矩阵被定义为了正方形矩阵。

```
#ifndef DATA
#define DATA "./Groebner/6_3799_2759_1953/"
#define COL 3799
#define ELE 2759
#define ROW 1953
#endif

#define mat_t unsigned int
#define mat_L 32

mat_t ele[COL][COL / mat_L + 1] = {0};
mat_t row[ROW][COL / mat_L + 1] = {0};

#ifndef ALIGN
mat_t ele_tmp[COL][COL / mat_L + 1] = {0}; // 消元子矩阵为正方形
mat_t row_tmp[ROW][COL / mat_L + 1] = {0};
#else
mat_t ele_tmp[COL][(COL / mat_L + 1) / 16 * 16 + 16] __attribute__((aligned(64))) = {0};
mat_t row_tmp[ROW][(COL / mat_L + 1) / 16 * 16 + 16] __attribute__((aligned(64))) = {0};
#endif

int main()
{
    // 读取数据集-消元子
    ifstream data_ele((string)DATA + (string) "1.txt", ios::in);
    int temp, header;
    string line;
    for (int i = 0; i < ELE; i++)
    {
        getline(data_ele, line);
        istringstream line_iss(line);
        line_iss >> header; // 先确定读入消元子的首元素
        ele[header][header / mat_L] += (mat_t)1 << (header % mat_L);
        // 将这一行消元子存进首元素对应的行，保证消元子矩阵为上三角形态，保证对角线的对应关系
        while (line_iss >> temp)
            ele[header][temp / mat_L] += (mat_t)1 << (temp % mat_L);
    }
```

```
38      data_ele.close();
39  }
```

2.2.2 串行算法

串行算法实现比较简单，代码如下：

```
1   void groebner(mat_t ele[COL][COL / mat_L + 1], mat_t row[ROW][COL / mat_L + 1])
2   {
3       // ele= 消元子，row= 被消元行
4       memcpy(ele_tmp, ele, sizeof(mat_t) * COL * (COL / mat_L + 1));
5       memcpy(row_tmp, row, sizeof(mat_t) * ROW * (COL / mat_L + 1));
6       for (int i = 0; i < ROW; i++)
7       { // 遍历行
8           for (int j = COL; j >= 0; j--)
9           { // 遍历列
10              if (row_tmp[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
11              { // 当前元素有值，需要进行处理
12                  if (ele_tmp[j][j / mat_L] & ((mat_t)1 << (j % mat_L)))
13                  { // 有对应消元子，对当前行进行异或
14                      for (int p = COL / mat_L; p >= 0; p--)
15                          row_tmp[i][p] ^= ele_tmp[j][p];
16                  }
17                  else
18                  { // 没有对应消元子，对当前行进行升格
19                      memcpy(ele_tmp[j], row_tmp[i], (COL / mat_L + 1) * sizeof(mat_t));
20                      break;
21                  }
22              }
23          }
24      }
25      // 省略：以稀疏矩阵格式输出 row_tmp，已与数据集中的参考结果对比一致
26  }
```

2.2.3 NEON/SSE 加速

NEON/SSE 的加速实现比较简单，为了方便处理，对于需要异或每一行的前面可以被寄存器宽度整除的元素进行 SIMD 加速运算，最后剩下来的几个元素则正常串行异或。

```
1   void groebner_simd(mat_t ele[COL][COL / mat_L + 1], mat_t row[ROW][COL / mat_L + 1])
2   {
3       // 省略：一堆无关的 for 和 if
```

```
                if (ele_tmp[j][j / mat_L] & ((mat_t)1 << (j % mat_L)))
                { // 当前元素有值，需要进行处理
                    for (int p = 0; p < COL / 128; p++)
                    { // 对于前面的 4n 个 u32，进行 simd 运算
                        row_i = vld1q_u32(row_tmp[i] + p * 4);
                        ele_j = vld1q_u32(ele_tmp[j] + p * 4);
                        vst1q_u32(row_tmp[i] + p * 4, veorq_u32(row_i, ele_j))
                    }
                    // 补足最后的几个 u32
                    for (int k = COL / 128 * 4; k <= COL / mat_L; k++)
                        row_tmp[i][k] ^= ele_tmp[j][k];
                }
    // 省略：一堆无关的 for 和 if
}
```

2.2.4 AVX 加速

AVX、AVX512 的实现与 NEON/SSE 的实现高度相似，需要进行的修改与普通高斯消去算法进行的修改差别不大，因此不再在报告中贴出代码，具体代码可以查看GitHub 仓库。

3 实验环境

3.1 X86 平台

由于这次的 X86 实验需要 AVX512 指令集的支持，而作者的笔记本电脑并不支持，因此使用了腾讯云的轻量服务器。服务器操作系统为 Debian 11 bullseye，Kernel 版本 x86_64 Linux 5.10.0-9-amd64；编译器为 gcc，版本 10.2.1 20210110；CPU 型号为 Intel Xeon Platinum 8255C @ 2x 2.494GHz，内存 1726MiB，KVM 虚拟化方式；CPU Cache 等信息如下 lscpu 命令输出所示：

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          2
On-line CPU(s) list:             0,1
Thread(s) per core:              1
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           85
```

```
Model name:                      Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz
Stepping:                        5
CPU MHz:                         2494.138
BogoMIPS:                        4988.27
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        8 MiB
L3 cache:                        35.8 MiB
NUMA node0 CPU(s):               0,1
```

3.2 ARM 平台

ARM 平台使用课程提供的鲲鹏云服务器，编译器 gcc，版本 9.3.1 20200408。

4 实验过程

4.1 普通高斯消去

首先生成测试数据，之后运行测试脚本。

```
g++ ./datagen.cpp -o datagen
./datagen
qsub gauss_sub.sh # 鲲鹏云服务器
bash gauss_timing.sh # 腾讯云轻量服务器
```

脚本会通过编译时宏定义来生成多个可执行文件，并将运行计时的结果写入 csv 文件中。

4.2 消元子模式的高斯消去

对于腾讯云 X86 上的实验，需要将 Groebner 数据集拷贝至脚本所在目录下。

```
qsub groebner_sub.sh # 鲲鹏云服务器
bash groebner_timing.sh # 腾讯云轻量服务器
```

这里使用的脚本可以在GitHub 仓库中找到，脚本中的编译命令使用的编译器均为 gcc，均使用了-march=native 选项和-w 选项。

5 实验结果

5.1 普通高斯消去

ARM 平台下普通高斯消去算法的运行时间与加速比随输入数据规模的变化如下图5.1和图5.2所示：

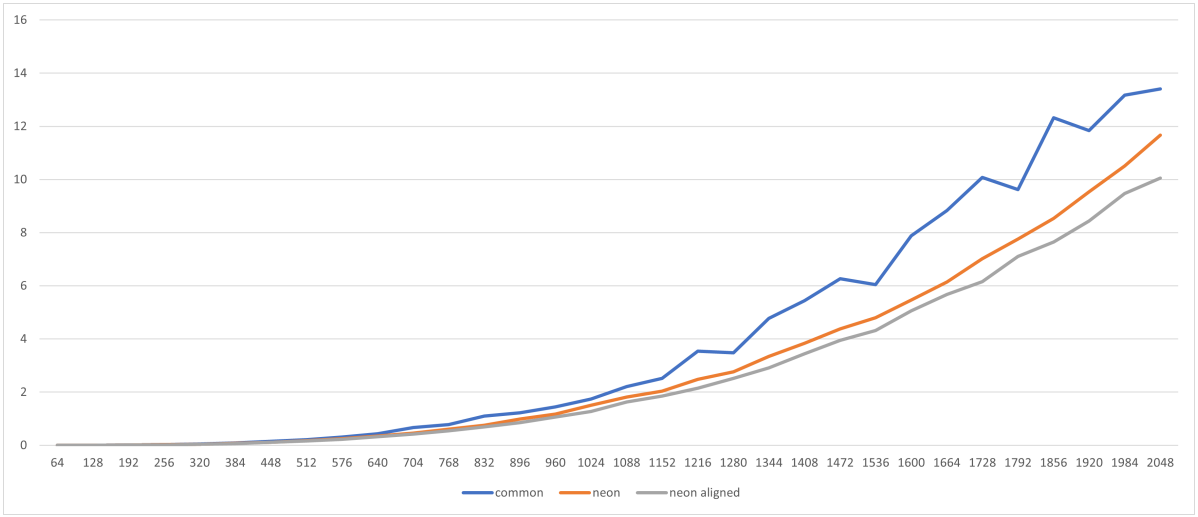

图 5.1: ARM 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

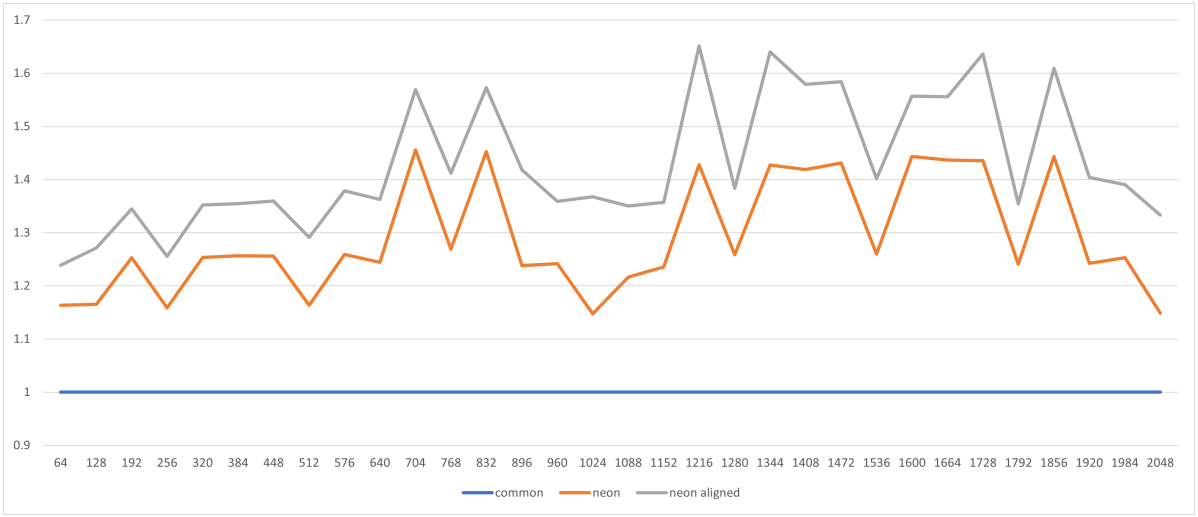

图 5.2: ARM 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

X86 平台下普通高斯消去算法的运行时间与加速比随输入数据规模的变化如下图5.3和图5.4所示：

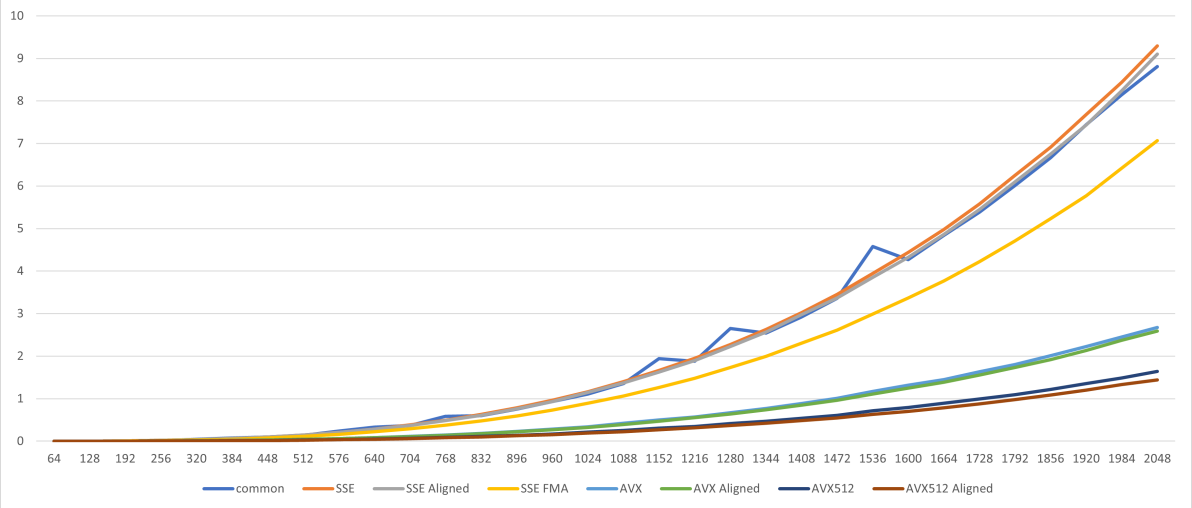

图 5.3: X86 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

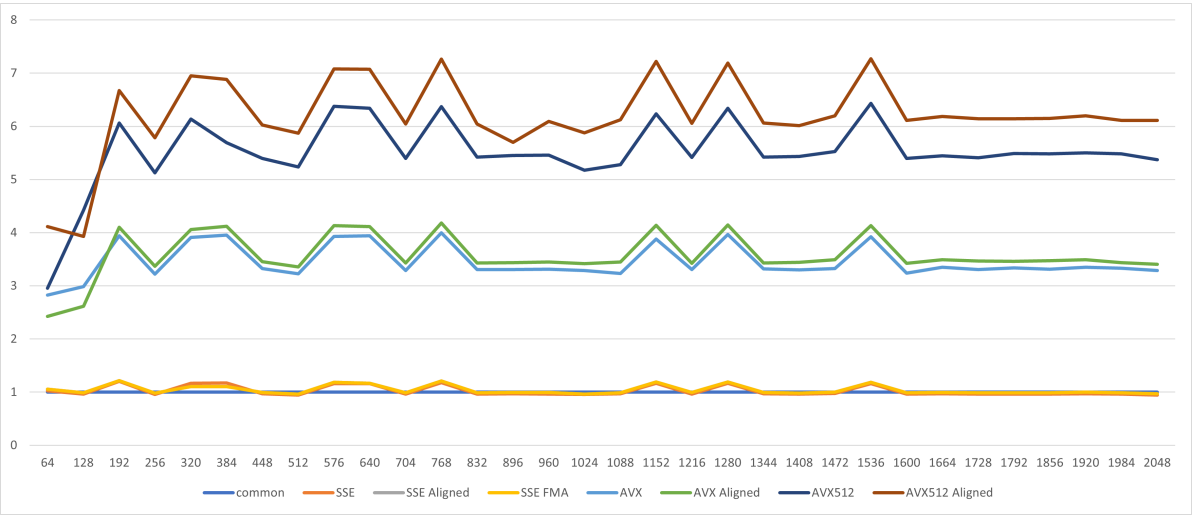

图 5.4: X86 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

5.2 消元子模式的高斯消去

ARM 平台下消元子模式高斯消去算法的运行时间与加速比随输入数据规模的变化如下表1和图5.5所示：

数据规模 \ 算法	common	NEON	common Aligned	NEON Aligned
3900	0.00001170	0.00000952	0.00001810	0.00001830
40386	0.00048108	0.00049586	0.00020015	0.00022082
125326	0.0009686	0.00079003	0.0004358	0.00048054
810822	0.0296091	0.025032	0.00594479	0.00580893
3965798	0.30704	0.188946	0.0510181	0.0353391
17900888	5.14359	3.78769	1.43977	0.945123
91633090	73.3229	48.1242	24.6904	14.6562

表 1: ARM 平台下特殊高斯消去算法运行时间 (单位:s)-输入数据规模（矩阵元素数量）

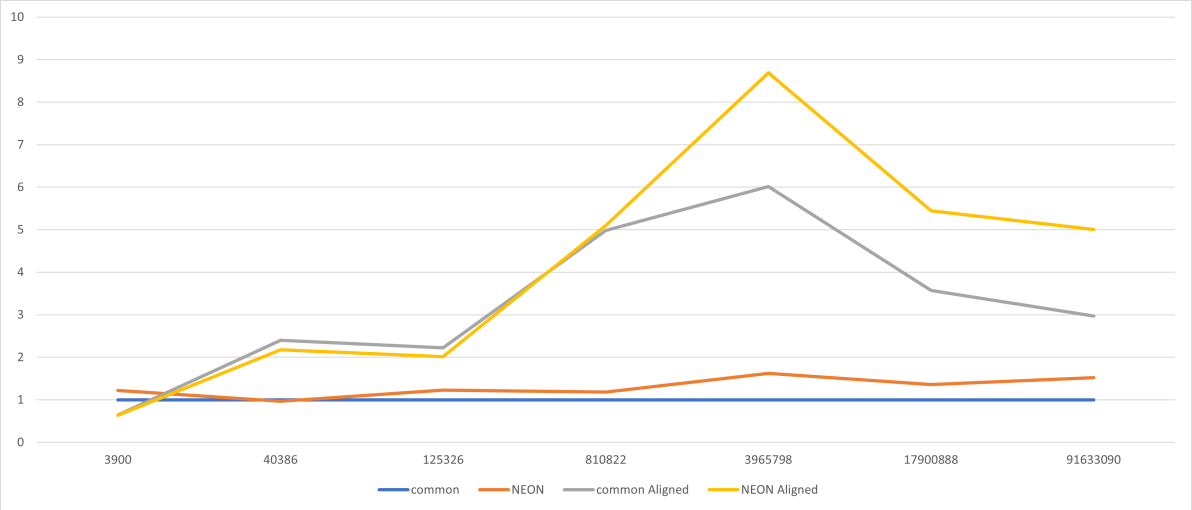

图 5.5: ARM 平台下特殊高斯消去算法加速比-输入数据规模（矩阵元素数量）

X86 平台下消元子模式高斯消去算法的运行时间与加速比随输入数据规模的变化如下表2、表3和图5.6所示：

数据规模 \ 算法	common	sse	avx	avx512
3900	0.000011	0.000007	0.000007	0.000007
40386	0.00040847	0.000377996	0.000392432	0.000448707
125326	0.000847442	0.000738246	0.000511157	0.000510796
810822	0.0264889	0.020673	0.015211	0.019706
3965798	0.205045	0.157969	0.0717809	0.0773024
17900888	3.53027	2.85676	1.29475	1.07147
91633090	46.0136	36.0756	14.8857	11.9147

表 2: X86 平台下特殊高斯消去不对齐算法运行时间 (单位:s)-输入数据规模（矩阵元素数量）

数据规模 \ 算法	common	sse	avx	avx512
3900	0.000017	0.000018	0.000013	0.000013
40386	0.000204125	0.000168772	0.000169869	0.000215989
125326	0.000384611	0.000369623	0.000382977	0.00039918
810822	0.00515151	0.00510496	0.00431197	0.00495711
3965798	0.0459342	0.0246126	0.0185454	0.0193652
17900888	1.14343	0.594648	0.397932	0.315436
91633090	16.3684	8.23317	5.05761	3.83209

表 3: X86 平台下特殊高斯消去对齐算法运行时间 (单位:s)-输入数据规模（矩阵元素数量）

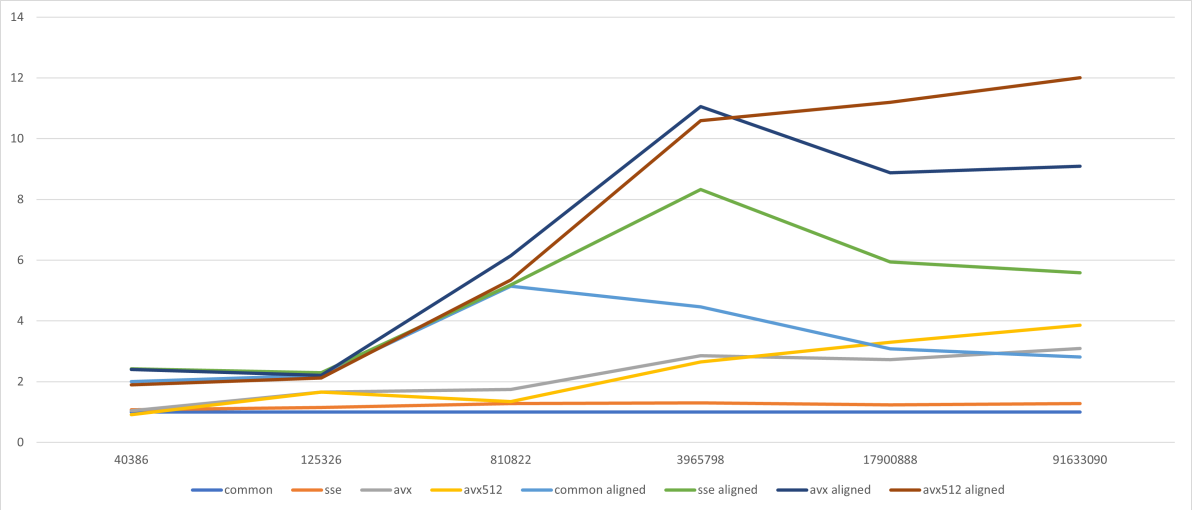

图 5.6: X86 平台下特殊高斯消去算法加速比-输入数据规模（矩阵元素数量）

6 结果分析

6.1 不同 SIMD 指令集对比

不难注意到，无论是在普通高斯消去（图5.2和图5.3）中，还是在消元子模式的高斯消去（图5.5和图5.6）中，当数据规模达到一定水平后，寄存器位数越高的 SIMD 指令集，其加速比也越高。这是因为寄存器位数决定了一次运算可以处理数据的的数量，实验中使用的数据类型为 unsigned int，AVX512 寄存器位数为 512，那么一次就可以处理 16 个数据，而 128 位的 SSE 和 NEON 就只能处理 4 个数据。

此外，SIMD 运算中大量的 Load/Store 也导致了一定的性能损失，在一次 Load/Store 只能处理四个数据的 SSE 下更加降低了加速比。ARM 平台上的 perf 事件计数表明，对齐的 NEON 加速特殊高斯消去算法的 35% 的 cycles 都耗费在了 vst1q 函数上，这或许能解释为何 NEON/SSE 加速算法的运行时间有时甚至长于普通算法。

6.2 数据对齐及 Cache 性能分析

对于特殊高斯消去算法，在 ARM 平台和 X86 平台上，我们不难发现，即使只是简单的将矩阵的定义从 $mat\_tele\_tmp[COL][COL/mat\_L+1]$; 改为 $mat\_tele\_tmp[COL][(COL/mat\_L+1)/16*16+16]\_\_attribute\_\_((aligned(64)))=0$; 也就是将二维数组的列数定义为 16 的整倍数，就可以给未使用 SIMD 的平凡算法带来最高 5 倍的加速比。

perf 性能测量后发现，对于未使用 SIMD 的平凡算法，修改前后 cache-misses 事件的估计数量从 39957488 降低到了 15547456，降幅达 61.1%；branch-misses 事件的估计数量从 16854480 降低到 5541842，降幅达 67.12%。

作者猜测，列数为 16 整倍数的二维数组可以更“整齐”的存放进 Cache 和内存中，而 Cache 和内存都是按字节块读取的，“整齐”的存放方式可以降低读取次数、提升性能。对齐的数据也能防止造成 Cache 的伪共享现象，进而充分利用 CPU 的超标量设计来提升性能。

此外，当需要在一个形如 $arr[M][N]$ 二维数组中寻址元素 $arr[i][j]$ 时，需要进行 $i*N+j$ 这样的操作，而如果 N 是 2 的幂，经过编译器的优化，可以更快的运行乘法，这样也就加速了寻址的过程。

而如果对于 SIMD 加速算法，对齐的数据更可以减少 Load/Store 的次数，带来更高的性能。

6.3 融合乘加所带来的性能影响

如图5.3和图5.4所示，SSE 加速在使用融合乘加的情况下有较明显的性能提升，查阅英特尔的 Intrinsics Guide[1] 得知，乘法指令 _mm_mul_ps，减法指令 _mm_sub_ps，和融合乘减 _mm_fnmadd_ps 指令延迟均为 4 个周期，CPI 均为 5，这样一来，使用融合乘加运算便可以在这一项上节约一半的时间，带来巨大的性能优势。

参考文献

[1] intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html Accessed April 24, 2022.

[2] intel/arm_neon_2_x86_sse_2022. https://github.com/intel/ARM_NEON_2_x86_SSE Accessed April 24, 2022.