南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验报告

SIMD 编程实验

李纡君

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 4 月 28 日

摘要

本实验中，我们对标准高斯消去法及其特定变体进行了深入分析和实施，采用了串行算法以及基于 NEON、SSE、AVX256 和 AVX512 指令集的 SIMD 并行化策略。实验不仅涵盖了各指令集的内存对齐优化技术，还对比了在不同计算环境下（包括个人笔记本、Intel DevCloud 和鲲鹏平台）的执行效率。此外，本研究详细探讨了算法的不同组成部分在向量化后的性能差异，并针对特定体系结构特征（如缓存机制）进行了优化以提升算法的可扩展性和效率。进一步地，通过采用 vtune 和 perf 等工具进行细致的性能剖析

关键字：SIMD，普通高斯消去，特殊高斯消去，neon，sse，avx256，avx512，对齐优化，体系结构，Intel devcloud

目录

一、 问题描述

(一) 普通高斯消去

从上到下，依次选取该行为操作行，在每轮操作中：

1. 首先，对操作行进行倍数处理操作，使首元为 1：

$$若首元为a_{ii}，则R_i \leftarrow \frac{1}{a_{ii}} R_i$$

其中，R_i 表示操作行。

2. 然后，用处理后的操作行去依次消掉后面各行的首元所在位置的元素，使得处理行首元以下全部为 0：

$$对于每一行j(其中j > i)，执行R_j \leftarrow R_j - a_{ji}R_i$$

其中，a_{ji} 是第 j 行在操作行首元列的元素。

3. 直到矩阵变为上三角矩阵为止。

(二) 特殊高斯消去

SIMD 优化特殊高斯消去的难点：

1. 实验数据给的是稀疏矩阵（并且存放于磁盘文件），问题包括如何读取数据并将稀疏矩阵转换为稠密矩阵，以及用什么数据类型存放稠密矩阵。
2. 受数据类型的限制，难点在于如何判断某行的消元子是否为空，以及如何获取被消元行的首项所在位置。
3. 设计算法以使其易于并行化的策略。
4. 设计分批操作，以及对批次覆盖不到的部分的特殊处理。

基于以上问题的算法设计分析：

- **数据读取与转换：**实现一个预处理步骤，将磁盘上的稀疏矩阵数据读入内存，并转换成适合进行 SIMD 操作的稠密格式。考虑使用二维数组或特定的数据结构如 `std::vector` 来存放稠密矩阵，确保可以高效地进行随机访问和计算。
- **消元子检测与获取：**采用有效的数据结构和检测机制来判定消元子的存在性，如使用标记数组或散列表跟踪每行的非零元素状态。同时，快速定位首项位置可以通过记录每行首个非零元素的索引来实现。
- **并行化设计：**利用数据的局部性原则，将矩阵划分为多个小块，每个块可以在不同的处理核上并行处理。同时考虑使用 SIMD 指令集直接对矩阵的行或列进行向量化操作，以提升处理速度。
- **分批与遗漏处理：**设计分批处理策略以适应内存限制，并为每批处理定义清晰的边界。对于批次覆盖不到的部分，设计补救措施，如迭代精细处理或后处理步骤，确保所有数据都得到正确处理。

二、 算法设计

(一) 普通高斯法

根据实验手册中提供的伪代码，设计了串行和并行的实验版本。两种算法的比较分析主要集中在嵌套循环的优化上。在串行算法中，一个双层循环用于执行更新

$$A[k][j] = \frac{A[k][j]}{A[k][k]}$$

，而三层循环用于计算

$$A[i][j] = A[i][j] - A[i][k] \times A[k][j]$$

。对于并行算法，通过使用一个含四个浮点数的向量寄存器来实现四路并行计算。在理想情况下，这种四路并行可以将执行时间缩减为原来的四分之一。然而，由于内存访问开销（需要将数据从内存加载到向量寄存器再存回），以及处理 $A[k][j]$ 对齐问题和不完全为四的情况下的额外开销，最终的优化效果可能不会达到四分之一的理想状态。

(二) 特殊高斯消去

1. 数据结构

对于稀疏矩阵的存储，我们使用 `unsigned int` 类型的二维数组。在这种方案中，每一行矩阵数据对应于数组的一行，每个 `unsigned int` 元素能存储 32 个 0 或 1 的数据。

存储策略如下：

- **消元子** (`Act[][]`):
  - **非空消元子**：首项的位置直接决定了消元子的行位置，例如首项位置为 200，则该消元子存储在 `Act[200][...]`。
  - **空消元子**:全 0 表示空消元子。为区分空和非空,每行数组的最后一个元素 `Act[row][last]` 设为 0 表示空行，设为 1 表示非空行。
- **被消元行** (`Pas[][]`):
  - **行号的处理**：被消元行在数组中的存储顺序与其在磁盘文件中的顺序相同。
  - **首项位置的存储**:为方便后续消元操作,首项位置在数组每行的最后一个元素 `Pas[row][last]` 中存储，并在读取磁盘时初始化，之后根据需要更新。

此方法确保了稀疏矩阵的高效存储及快速访问，特别是在进行矩阵消元等操作时，可以直接定位到关键行和列，从而提高计算效率。

2. 算法步骤

特殊高斯消元法是处理稀疏矩阵的一种有效方法，特别是当矩阵以压缩形式存储时。下面是这一算法的具体步骤:

1. **初始化**：将矩阵转化为二维 `unsigned int` 数组形式，其中每 32 位的一组 `unsigned int` 存储对应的 32 个矩阵元素。
2. **选择主元**：从 `Act` 数组中选择非空消元子作为主元。如果 `Act[row][last]` 为 1，则该行非空，可用作消元。

3. **消元过程**:

   (a) 对于每一个被消元行，使用 `Pas[row][last]` 确定首项位置。

   (b) 使用选定的主元行对所有被消元行进行消元操作，即通过异或操作将主元行首项对齐的部分清零。

   (c) 更新被消元行在 `Pas` 数组中的表示和首项位置。

4. **检查和调整**：每次消元后，检查并更新消元子的状态，如果某消元子行变为空（即全部为零），则将 `Act[row][last]` 设置为 0。

5. **重复步骤**：重复以上步骤直至所有行被处理，或无法继续找到有效的非空消元子。

3. 并行部分实现

因为该算法有很多的判断语句，所以并非所有部分都能够完美使用 SIMD 并行，在本算法中，用来并行优化的核心代码是一个三重循环的数组异或操作，使用 4 路 uint 向量寄存器，对该循环进行并行操作。

三、 基于 neon 算法实现

(一) 普通高斯消去

1. normal

```
    void f_ordinary() {
    for (int k = 0; k < n; k++) {
        for (int j = k + 1; j < n; j++) {
            A[k][j] = A[k][j] * 1.0 / 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. Cache 优化

普通高斯消去 cache 优化代码

```
void f_ordinary_cache() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            B[j][i] = A[i][j];
            A[i][j] = 0; // 相当于原来的 A[i][k] = 0
```

```
        }
    }
    for (int k = 0; k < n; k++) {
        for (int j = k + 1; j < n; j++) {
            A[k][j] = A[k][j] * 1.0 / 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] - B[k][i] * A[k][j];
            }
        }
    }
}
```

普通高斯消去——并行 + cache 优化

```
void f_pro_cache_revised() {
    // Preprocessing to create an upper triangular matrix in B from A
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            B[j][i] = A[i][j];
            A[i][j] = 0;
        }
    }

    // Normalize the diagonal elements of A to 1 and scale the upper part
        using SIMD
    for (int k = 0; k < n; k++) {
        float32x4_t diagonal = vmovq_n_f32(A[k][k]);
        int next_col = k + 1;
        for (; next_col + 3 < n; next_col += 4) {
            float32x4_t current = vld1q_f32(&A[k][next_col]);
            current = vdivq_f32(current, diagonal);
            vst1q_f32(&A[k][next_col], current);
        }
        for (; next_col < n; next_col++) {
            A[k][next_col] = A[k][next_col] / A[k][k];
        }
        A[k][k] = 1.0;

        // Update the lower rows
        for (int i = k + 1; i < n; i++) {
            float32x4_t multiplier = vmovq_n_f32(B[k][i]);
            int col_start = k + 1;
            for (; col_start + 3 < n; col_start += 4) {
                float32x4_t upper_row = vld1q_f32(&A[k][col_start]);
                float32x4_t lower_row = vld1q_f32(&A[i][col_start]);
```

```
                float32x4_t product = vmulq_f32(upper_row, multiplier);
                lower_row = vsubq_f32(lower_row, product);
                vst1q_f32(&A[i][col_start], lower_row);
            }
            for (; col_start < n; col_start++) {
                A[i][col_start] -= B[k][i] * A[k][col_start];
            }
        }
    }
}
```

这段代码是一个优化后的高斯消元算法，利用 NEON SIMD 指令集加速计算。首先，代码对矩阵 A 进行转置存储到 B，然后使用 float32x4_t 类型进行向量化操作，实现四个元素的同时处理以提高性能。主要过程包括矩阵的标准化、除法和消去步骤，通过并行化处理向量化数据，从而加速整个矩阵的消元过程。

3. 普通高斯消去 neon 并行优化

并行优化的实现基于 neon 架构，使用 float 的向量寄存器 float32x4 t、load 操作 vld1q f32、store 操作 vst1q f32 等等。

并行算法优化见下:

```
// Auxiliary function to divide elements of a row by a scalar using SIMD
void simd_divide_row_elements(float* row, float divisor, int start, int
    length) {
    float32x4_t vdivisor = vmovq_n_f32(divisor);
    int i;
    for (i = start; i + 4 <= length; i += 4) {
        float32x4_t elements = vld1q_f32(&row[i]);
        elements = vdivq_f32(elements, vdivisor);
        vst1q_f32(&row[i], elements);
    }
    for (; i < length; i++) {
        row[i] = row[i] / divisor;
    }
}

// Auxiliary function to update matrix rows using SIMD
void simd_update_rows(float **matrix, int current, int start, int size, float
     factor) {
    float32x4_t vfactor = vmovq_n_f32(factor);
    for (int i = current + 1; i < size; i++) {
        float32x4_t* row_i = (float32x4_t*) &matrix[i][start];
        float32x4_t* row_current = (float32x4_t*) &matrix[current][start];
        for (int j = 0; j < (size - start) / 4; j++) {
            float32x4_t vakj = vld1q_f32((float*) &row_current[j]);
            float32x4_t vaij = vld1q_f32((float*) &row_i[j]);
            float32x4_t product = vmulq_f32(vakj, vfactor);
            vaij = vsubq_f32(vaij, product);
```

```
            vst1q_f32((float*) &row_i[j], vaij);
        }
    }
}

void f_pro_cache_updated() {
    for (int k = 0; k < n; k++) {
        // Normalize the k-th row elements using SIMD
        simd_divide_row_elements(A[k], A[k][k], k + 1, n);

        // Set the diagonal element to 1
        A[k][k] = 1.0;

        // Update subsequent rows based on the k-th row
        for (int i = k + 1; i < n; i++) {
            float factor = A[i][k];
            simd_update_rows(A, k, k + 1, n, factor);
            A[i][k] = 0;
        }
    }
}
```

4. 普通高斯消去 neon 并行优化——对齐优化

由于高斯消去计算过程中，第 k 步消去的起始元素是变化的，从而导致距 16 字节边界的偏移是变化的，因此我们可以通过修改代码，手动令第 k 步消去的起始元素对齐。对齐代码如下:

```
void inline process_row_segment(float* row, float divisor, int start, int end
    ) {
    int i = start;
    while ((i % 4) != 0 && i < end) {
        row[i] = row[i] / divisor;
        i++;
    }
    float32x4_t vdivisor = vmovq_n_f32(divisor);
    for (; i + 4 <= end; i += 4) {
        float32x4_t elements = vld1q_f32(&row[i]);
        elements = vdivq_f32(elements, vdivisor);
        vst1q_f32(&row[i], elements);
    }
    for (; i < end; i++) {
        row[i] = row[i] / divisor;
    }
}

void inline update_row(float* row, float* reference, float factor, int start,
     int end) {
    int i = start;
```

```
    while ((i % 4) != 0 && i < end) {
        row[i] -= reference[i] * factor;
        i++;
    }
    float32x4_t vfactor = vmovq_n_f32(factor);
    for (; i + 4 <= end; i += 4) {
        float32x4_t ref = vld1q_f32(&reference[i]);
        float32x4_t current = vld1q_f32(&row[i]);
        float32x4_t product = vmulq_f32(ref, vfactor);
        current = vsubq_f32(current, product);
        vst1q_f32(&row[i], current);
    }
    for (; i < end; i++) {
        row[i] -= reference[i] * factor;
    }
}

void f_pro_cache_condensed() {
    for (int k = 0; k < n; k++) {
        process_row_segment(A[k], A[k][k], k + 1, n);
        A[k][k] = 1.0;

        for (int i = k + 1; i < n; i++) {
            update_row(A[i], A[k], A[i][k], k + 1, n);
            A[i][k] = 0.0;
        }
    }
}
```

在实施本实验中，首先需要确保二重循环和三重循环中的数据对齐，具体来说，检查 $k \times n + j$ 是否为 4 的倍数，即数组 $A[k][j]$ 是否按 16 字节对齐。如果未对齐，则先行执行串行代码直至满足对齐条件后，再执行并行操作。在实验过程中发现，原对齐策略存在问题，影响了实验结果的优化效果。因此，本文进行了以下改进：

1. **二维数组的连续存储**：最初使用的是动态分配每行的 A = new float*[] 方法，这种方式虽然每行内存是连续的，但行与行之间并不连续。这导致在缓存访问时增加了内存访问开销。因此，改为使用全局数组 $A[n][n]$ 的方式直接在一个连续的内存块中构造二维数组，确保整体的连续性。

2. **对齐位置的计算**：基于上述讨论，检查对齐位置时主要考察 $k \times n + j$ 是否为 4 的倍数。实验中还探索了在三重循环中同时对 $A[k][j]$ 和 $A[i][j]$ 进行对齐，发现两者在性能上的差异微小，故最终决定在代码中仅展示对 $A[k][j]$ 进行对齐的方法。

(二) 特殊高斯消去

1. 初始化 (由稀疏矩阵构造稠密矩阵)

下面以消元子的初始化为例，被消元行的初始化与消元子只差别在每行最后一个元素（被消元行存的是首项，消元子存的是否为空）

特殊高斯消去——初始化数组

```
unsigned int a;
char fin[10000] = {0};
ifstream infile("act.txt");
int index;

while (infile.getline(fin, sizeof(fin))) { // 从文件中提取行
    std::stringstream line(fin);
    int biaoji = 0;

    while (line >> a) { // 从行中提取单个的数字
        if (biaoji == 0) {
            index = a; // 取每行第一个数字为行标
            biaoji = 1;
        }
        int k = a % 32, j = a / 32;
        int temp = 1 << k;
        Act[index][262 - j] += temp;
        Act[index][263] = 1; // 记录消元子是否为空，为空记0，否则记1
    }
}
```

2. 串行实现

特殊高斯消去——串行

```
void f_ordinary() {
    int i; // 每轮处理8个消元子，范围：首项在i-7到i
    for (i = 8398; i - 8 >= -1; i -= 8) {
        // 遍历被消元行，寻找首项在范围内的行
        for (int j = 0; j < 4535; j++) {
            while (Pas[j][263] <= i && Pas[j][263] >= i - 7) {
                int index = Pas[j][263];
                if (Act[index][263] == 1) { // 消元子不为空
                    for (int k = 0; k < 263; k++) // 与消元子异或
                        Pas[j][k] = Pas[j][k] ^ Act[index][k];

                    // 更新Pas[j][263]存的首项值，根据新的首项值决定是否退出循环
                    int num = 0, S_num = 0;
                    for (num = 0; num < 263; num++) {
                        if (Pas[j][num] != 0) {
                            unsigned int temp = Pas[j][num];
                            while (temp != 0) {
                                temp = temp >> 1;
                                S_num++;
                            }
                            S_num += num * 32;
```

```
                            break;
                        }
                    }
                    Pas[j][263] = S_num - 1;
                } else { // 消元子为空，Pas[j]行升格为消元子
                    for (int k = 0; k < 263; k++)
                        Act[index][k] = Pas[j][k];
                    Act[index][263] = 1; // 设置消元子标志非空
                    break;
                }
            }
        }
    }
    // 处理剩余部分，为节省空间，这部分省略；具体操作与上面相同
    for (i = i + 8; i >= 0; i--) {
        // 省略的部分
    }
}
```

3. 并行优化部分

分析上面的串行算法，有一些控制流无法进行 SIMD 优化，选取可以并行优化的部分——三重循环中的异或操作

对该部分的 neon 并行优化代码如下：

特殊高斯消去——neon 并行:

```
// ************* 并行优化部分 *************
int k;
for (k = 0; k + 4 <= 263; k += 4) {
    uint32x4_t vaPas = vld1q_u32(&(Pas[j][k])); // 加载Pas
    uint32x4_t vaAct = vld1q_u32(&(Act[index][k])); // 加载Act
    vaPas = veorq_u32(vaPas, vaAct); // 向量按位异或
    vst1q_u32(&(Pas[j][k]), vaPas); // 存储
}
for (; k < 263; k++) { // 处理剩余部分
    Pas[j][k] = Pas[j][k] ^ Act[index][k];
}
// ************* 并行优化部分 *************
```

四、 鲲鹏实验结果及分析（其他优化结果见下面章节）

(一) 普通高斯消去

1. 问题规模

由实验结果可知：

1. 并行算法相比于串行算法有着显著的提速，但是加速比约为 1.5 左右。

表 1: 普通高斯消去法（不同问题规模）实验结果

问题规模	200	500	1000	2000	3000
串行时间/ms	19.12	302.53	2435.64	19621.5	51209.7
并行时间/ms	12.85	198.23	1593.75	12840.3	43920.4
加速比	1.516	1.527	1.530	1.527	1.165

2. 不同问题规模的加速比较为一致（推测当问题规模为 $n = 3000$ 时较低的加速比可能是因为访存、时间测量等原因造成的，这是由于在 $n = 3000$ 的前后加速比均维持在 1.5 左右，并没有出现显著的数学关系）。

加速比未达到 4 的可能原因包括：

1. **访存开销**：向量寄存器的 load 和 store 操作导致的开销。特别是当访存地址没有对齐时，可能会进一步影响速度。
2. **未向量化部分的开销**：由于处理的数据规模不总是 4 的整数倍，存在无法并行化的部分，这些部分的存在会降低整体的加速比。
3. **非并行部分的开销**：例如 $A[k][k]$ 和 $A[i][k]$ 的赋值操作等不可并行化的代码段，也会对总体加速比产生影响。
4. **其他开销**：比如函数调用等额外开销。

2. 对齐操作对实验结果的影响

表 2: 普通高斯消去法（对于操作数的比）实验结果

问题规模	200	500	1000	2000	3000
串行时间/ms	19.24	301.54	2434.85	19620.5	51208.1
并行时间/ms	12.75	197.28	1592.39	12836.9	43918.7
并行 + 对齐时间/ms	11.85	181.16	1441.02	11519.3	40220.0

经结果验证得出，本文采用的对齐策略 (见“算法实现”一节) 在不同规模下对实验性能的提升都有一定效果。

3. 对不同部分向量化的对比结果

在普通高斯消去的 neon 并行版本中，对两个循环做了优化，为了探究这两个优化片段分别对总体性能的提升程度，设计对比实验，结果如如表 3 所示：

从表中分析得出以下结论：

1. 当我们对除法的双重循环进行并行化时，执行时间并不总是比串行执行短，有时甚至更长。这可能是因为：
   - 双重循环在整个算法中占的比重相对较小。
   - 并行化引入的内存访问（加载和存储）开销可能抵消了并行处理的好处。

   当并行化导致的开销超过其性能优势时，这就可能导致整体性能下降。这表明我们在进行并行优化时，必须谨慎考虑优化带来的内存访问成本。

表 3: 通道计算耗时（不同阵列大小）实验结果

问题规模	200	500	1000	2000	3000
串行时间/ms	19.284	301.546	2434.87	19620.7	51208.3
并行（静态分配）时间/ms	12.712	197.319	1592.47	12837.6	43919.8
并行（动态分配任务）时间/ms	19.261	302.944	2430.39	19622.4	54467.4
并行（动态分配任务优化）时间/ms	12.762	197.923	1604.71	12853.1	43779.8

2. 另一方面，当仅对三重循环的消去步骤进行并行化时，观察到的执行时间与完全并行化非常相近，甚至有轻微的增加，这表明：

   - 三重循环是高斯消元算法中决定性能的关键部分。

   进一步分析这两种循环的复杂性我们可以看到：

- 双重循环中，每次迭代都会进行一次内存加载、一次向量除法运算和一次内存存储，总共重复了 n^2 次。
- 三重循环中，每次迭代都包括两次内存加载、两次运算和一次内存存储，共需要进行 n^3 次迭代。

因此，由于三重循环的复杂度和执行操作的数量都远高于双重循环，它在算法的整体性能中起着决定性的作用。

(二) 特殊高斯消元法

表 4: 转录音频消耗时间（不同问题规模）实验结果

矩阵列数	9000	24000	40000
串行时间/ms	100.34	230.21	480.47
并行时间/ms	85.76	220.39	440.52

在进行并行计算时，可以观察到并行处理带来了一定程度的加速效果。然而，特殊高斯算法中包含大量的控制流结构，这些结构限制了并行化的可能性。因此，这部分的代码无法有效并行化，导致整体的加速比不如传统的高斯消元法那么显著。

五、 探究不同指令集和平台对实验的影响

(一) 普通消元法

1. 实验结果分析

SSE 指令集、AVX-256、AVX-512 代码见 github 链接, 直接分析实验结果 在进行跨平台性能分析中，类似于特殊高斯算法的现象得以重现：vs2019 环境的执行效率相较于 Intel devcloud

表 5: 算法执行时间（不同指令集和编译器）对比表

测试样例	300	500	1000	2000	3000
SSE (dev) / ms	4.36	7.91	80.62	517.78	2542.37
AVX-256 (dev) / ms	3.84	6.14	57.55	471.40	2317.17
AVX-512 (dev) / ms	1.60	3.87	52.49	452.24	2111.37
SSE (vs2019) / ms	28.96	360.84	643.47	3486.73	12030.43
AVX-256 (vs2019) / ms	6.31	75.53	377.57	1994.64	6939.02

明显较低。这可能归因于 vs2019 在本地环境中受限于更多的系统级别约束，而 devcloud 由于其专业化的资源配置和任务调度优化，提供了更为高效的执行环境。

细究不同指令集的计算效能，展现出一致性增进的模式：无论是在 vs2019 还是在 devcloud 环境中，sse、avx-256、avx-512 指令集在相同计算负载下显示出递增的计算速度，与它们的并行度呈正相关——分别为四、八、十六通道并行。特别是在小规模数据集上，执行时间与并行通道数的比率接近一致，指向并行度的有效利用；而在大规模数据集上，内存访问和缓存效率的影响逐渐占据主导，暗示在算法设计时应深入考量数据规模与并行度之间的关系。这一洞察强调了并行计算架构在高效算法设计中的核心地位。

2. SSE 指令对齐

```
// ********************************** 并行优化部分
__m128 va, vt, vx, vaij, vaik, vakj;

void f_sse_alignment() {
    for (int k = 0; k < n; k++) {
        vt = _mm_set_ps(A[k][k], A[k][k], A[k][k], A[k][k]);
        int j = k + 1;
        while ((k * n + j) % 4 != 0) {
            // 对齐
            A[k][j] = A[k][j] * 1.0 / A[k][k];
            j++;
        }
        for (; j + 4 <= n; j += 4) {
            va = _mm_load_ps(&(A[k][j]));
            va = _mm_div_ps(va, vt);
            _mm_store_ps(&(A[k][j]), va);
        }
        for (; j < n; j++) {
            A[k][j] = A[k][j] * 1.0 / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < n; i++) {
            vaik = _mm_set_ps(A[i][k], A[i][k], A[i][k], A[i][k]);
            int j = k + 1;
            while ((i * n + j) % 4 != 0) {
                // 对齐
                A[i][j] = A[i][j] - A[k][j] * A[i][k];
```

```
                j++;
            }
            for (; j + 4 <= n; j += 4) {
                vakj = _mm_load_ps(&(A[k][j]));
                vaij = _mm_load_ps(&(A[i][j]));
                vx = _mm_mul_ps(vakj, vaik);
                vaij = _mm_sub_ps(vaij, vx);
                _mm_store_ps(&A[i][j], vaij);
            }
            for (; j < n; j++) {
                A[i][j] = A[i][j] - A[k][j] * A[i][k];
            }
            A[i][k] = 0.0;
        }
    }
}
// ********************************** 并行优化部分
```

表 6: SSE 对齐与不对齐实验结果

测试规模	1000	2000	50000
SSE 不对齐/ms	1064.99	2022.64	6020.36
SSE 对齐/ms	730.38	1501.25	4036.65

实验结果表明对齐代码均比不对齐代码运行时间更短，效率更高。

(二) 特殊高斯消元法

1. SSE 指令集 (x86)

```
// ********************************** 并行优化部分
int k;
for (k = 0; k + 4 <= Num; k += 4){
    // Pas[j][k] = Pas[j][k] ^ Act[index][k];
    __m128 va_Pas = _mm_loadu_ps((float *)&(Pas[j][k]));
    __m128 va_Act = _mm_loadu_ps((float *)&(Act[index][k]));
    va_Pas = _mm_xor_ps(va_Pas, va_Act);
    _mm_store_ss((float *)&(Pas[j][k]), va_Pas);
}
for (; k < Num; k++){
    Pas[j][k] = Pas[j][k] ^ Act[index][k];
}
// ********************************** 并行优化部分
```

2. AVX-256 指令集 (x86)

```
// ********************************* 并行优化部分
int k;
for (k = 0; k + 8 <= Num; k += 8){
    __m256 va_Pas2 = _mm256_loadu_ps((float *)&(Pas[j][k]));
    __m256 va_Act2 = _mm256_loadu_ps((float *)&(Act[index][k]));
    va_Pas2 = _mm256_xor_ps(va_Pas2, va_Act2);
    _mm256_storeu_ps((float *)&(Pas[j][k]), va_Pas2);
}
for (; k < Num; k++){
    Pas[j][k] = Pas[j][k] ^ Act[index][k];
}
// ********************************* 并行优化部分
```

3. AVX-512 指令集 (x86)

```
// ***************************************** 并行优化部分
int k;
for (k = 0; k + 16 <= Num; k += 16) {
    va_Pas3 = _mm512_loadu_ps((float *)&(Pas[j][k]));
    va_Act3 = _mm512_loadu_ps((float *)&(Act[index][k]));
    va_Pas3 = _mm512_xor_ps(va_Pas3, va_Act3);
    _mm512_storeu_ps((float *)&(Pas[j][k]), va_Pas3);
}
for (; k < Num; k++) {
    Pas[j][k] = Pas[j][k] ^ Act[index][k];
}
// ***************************************** 并行优化部分
```

4. 结果分析

表 7: 算法执行时间对比（不同指令集和编译器）

测试样例	test1	test2
SSE (dev) / ms	120.05	267.24
AVX-256 (dev) / ms	125.57	263.34
AVX-512 (dev) / ms	136.08	309.119
SSE (vs2019) / ms	372.54	981.81
AVX-256 (vs2019) / ms	322.27	771.66

test1 矩阵列数 23045，非零消元子 18748，被消元行 14325

test2 矩阵列数 37960，非零消元子 29304，被消元行 14921

在对比不同平台的执行速度时，可以观察到在 `vs2019` 上运行的效率明显低于在 `Intel devcloud` 上。进一步比较不同的指令集，我们发现在 `vs2019` 环境中，AVX 指令由于其 8 路并行能力，相较于 SSE 的 4 路并行提供了更好的性能。然而，当运行于 `Intel devcloud` 平台

时，这种优势并未显现。高并行度的指令集如 AVX 并没有展现出预期的性能提升。理论分析表明，这可能是因为特殊高斯消元算法中控制流的广泛存在。算法中真正可供并行处理的部分并不多，随着并行度的增加，向量化的开销也随之增大，因为需要处理更长的向量。同时，不适合并行处理的部分也增多，特别是当考虑到模 4、模 8 和模 16 的操作时，这可能导致尽管并行度提高了，但实际性能并没有得到相应的提升。

六、 profiling

1. vtune

表 8: 性能指标对比

优化	CPU Time	CPI Rate	Instructions Retired	Clockticks
普通平均算法	0.748s	0.498	6,348,000,000	3,148,000,000
SSE 优化	0.447s	0.648	2,862,000,000	1,852,000,000
SSE 优化 + 调整	0.448s	0.632	2,868,000,000	1,822,000,000
AVX-256 优化	0.252s	0.708	1,477,000,000	1,038,000,000

当深入分析指令集并行化的性能影响时，可以明确地观察到随着并行通道数量的增加，总体的循环迭代次数和指令执行量均有所下降，进而直接减少了 CPU 时间的消耗，与先行的实验趋势一致。不过，这种并行化也导致 CPI 的提升，这表明了更多并行通道的单指令执行周期有所增长。这可能源于并行计算中固有的同步开销以及对计算资源的动态分配机制。

这一现象揭示了处理器架构设计中并行度与单指令效率之间的微妙平衡，提示在增加计算密集型任务的吞吐量时，可能会对指令流水线的效率产生非预期影响。因此，处理器架构师在设计指令集时需要细致评估并行化对指令级并行处理（ILP）的影响。同时，软件开发者在面对高度并行的指令集时，需通过算法优化来抵消同步与调度所带来的性能开销，从而确保并行化技术带来的性能提升得以充分实现。

七、 源代码

https://github.com/uJunLI/NKU-Parallel_programming//lab2