南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

SIMD 编程实验

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 4 月 12 日

摘要

本实验基于两种高斯消去算法，进行 **SIMD** 并行实验。分别对**普通高斯消去**和**特殊高斯消去**两种算法，实现了串行算法和 **neon**、**sse**、**avx256**、**avx512** 指令集的并行优化，并且拓展分析了基于各指令集的**对齐优化**、**不同平台**（个人笔记本和 **Intel devcloud**、鲲鹏）的对比、对算法不同部分分别向量化的对比，实现了基于**体系结构**（cache 相关）的拓展性优化算法以及其他优化策略等。最后给出了基于 vtune 和 perf 的 **profiling** 分析。

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、 问题分析

(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：

首先，对操作行进行倍数处理操作，使首元为 1；

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元素，使得处理行首元以下全为 0；

直到矩阵变为三角矩阵为止。

(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），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，如何判断某行的消元子是否为空，如何获取被消元行的首项所在位置。

3) 如何设计算法使得其易于并行化。

4) 如何设计分批操作，以及批次覆盖不到的部分的操作。

- 基于以上问题分析思路，进行了下面的算法设计。

二、 算法设计与分析

(一) 普通高斯消去

根据实验指导书的伪代码设计串行和并行实验。（不再赘述）

对比分析串行和并行算法，并行优化主要体现在双重循环和三重循环部分：

串行算法：在一个二重循环里实现 // A[k][j] = A[k][j] / A[k][k] // ；在三重循环实现 // A[i][j] = A[i][j] - A[i][k] * A[k][j] //

并行算法：用一个四浮点数的向量寄存器来四路并行的进行运算操作，理想情况下，四路并行会让时间减少到原来的 1/4，但是由于访存开销（并行时需要把数据从内存 load 到向量寄存器，再 store 回去）以及每次读取 A[k][j] 难对齐、不满足 4 路的单独开销等，预计最后的优化效果几乎无法达到 1/4。

(二) 特殊高斯消去

1. 数据结构的设计

将稀疏矩阵存为一个 unsigned int 类型的二维数组，数组的一行代表矩阵的一行，矩阵每 32 个 0/1 对应于数组的一个 unsigned int（32 位）元素。

但消元子和被消元行的存储方式存在差异：

- 消元子：二维数组 Act[ ][ ]

1. 非空消元子的处理：非空消元子的首项所在位置决定了该消元子所在行，例如：首项为 200 的消元子存放于 Act[200][...]（为了之后消元方便）

2. 空消元子的处理：本文中将空消元子存为全 0，但为了区分空与非空，在数组每行的最后设置一个元素 Act[row][last]，空行为 0，非空消元子为 1。

- 被消元行：二维数组 Pas[ ][ ]

1. 行号的处理：被消元行在数组的每行存储顺序与其在磁盘文件一致。

2. 首项所在位置的存储：由于之后消元时，每次都要提取被消元行首项所在位置，所以也在数组每行的最后设置一个元素 Pas[row][last]，用于存放被消元行的首项，在读取磁盘稀疏矩阵时就做好初始化，之后逐步更新。

2. 算法步骤

1) 逐批次读取消元子 Act[ ]。

2) 对当前批次中每个被消元行 Pas[ ]，检查其首项（Pas[row][last]）是否有对应消元子；若有，则将与对应消元子做异或并更新首项（Pas[row][last]），重复此过程直至 Pas[row][last] 不在范围内。

3) 运算中，若某行的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，copy 到数组 Act 的对应行，并设标志位为 1 表示非空，然后结束对该被消元行的操作。

运算后若每行的首项不在当前批次覆盖范围内，则该批次计算完成；

4) 重复上述过程，直至所有批次都处理完毕。

3. 并行部分的设计

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、 算法实现（此部分仅做了基于 neon 的讨论，更多优化见下面章节）

（一） 普通高斯消去

1. 未对齐操作

首先需要初始化一个适合于高斯消去的矩阵，实现方法参考了指导书。

串行算法实验指导书已经给出，不再赘述。

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：

普通高斯消去 neon 并行优化

```
for (int k = 0; k < n; k++){
        float32x4_t vt=vmovq_n_f32(A[k][k]);  int j;
            for (j = k + 1; j+4 <= n; j+=4){
                float32x4_t va=vld1q_f32(&(A[k][j]));
                    va= vdivq_f32(va,vt);
                    vst1q_f32(&(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++){
                    float32x4_t vaik=vmovq_n_f32(A[i][k]);
                    for (int j = k + 1; j+4 <= n; j+=4){
                            float32x4_t vakj=vld1q_f32(&(A[k][j]));
                            float32x4_t vaij=vld1q_f32(&(A[i][j]));
                            float32x4_t vx=vmulq_f32(vakj,vaik);
                            vaij=vsubq_f32(vaij,vx);
                            vst1q_f32(&A[i][j], vaij);
                    }
                    for(; j<n; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
                    A[i][k] = 0;
                }
        }
```

2. 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：

普通高斯消去 neon 并行优化——对齐代码

```
    for(int k = 0;k < n; k++){
        float32x4_t vt = vmovq_n_f32(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 = vld1q_f32(&A[k][j]);
            va = vdivq_f32(va,vt);
            vst1q_f32(&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 = vmovq_n_f32(A[i][k]);
            int j = k + 1;
            while((k * 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 = vld1q_f32(&A[k][j]);
                vaij = vld1q_f32(&A[i][j]);
                vx = vmulq_f32(vakj,vaik);
                vaij = vsubq_f32(vaij,vx);
                vst1q_f32(&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;
    }}
```

如代码所示，在执行二重循环和三重循环之前，先检查 k*n+j 是否为 4 的倍数，即 A[k][j] 是否 16 字节对齐，若未对齐，则先执行串行代码，等到对齐了再开始并行。

具体做实验时，发现对齐算法中存在一些问题会影响实验结果，本文更改了以下问题，最后使得对齐操作可以达到优化效果：

1) 二维数组 A[ ][ ]：开始时，使用的是 A = new float*[ ] 并用循环为每个 A[x] 再 new 一块内存，然而，在这种方法下，A[x]（即二维数组的每行）虽然是一块连续的空间，但行与行之间不连续。这就导致了在 cache 访存二维数组时，要进行不连续的内存访问，增加了访存开销。因此，后来改进使用了全局变量 A[n][n] 直接构造的方法，保证二维数组 A[][] 在一整块内存空间里。

2) 基于 1) 的讨论，我在寻找 A[k][j] 的对齐位置时，探究的是 k*n+j 是否为 4 的倍数，这是从整个二维数组 A[ ][ ] 的起始地址开始计算的。

在对三重循环的对齐操作时，本文分别对对齐 k*n+j（A[k][j]）和对齐 i*n+j（A[i][j]）进行了实验，结果发现，两者的差异非常小，可以忽略，所以最后本文代码在三重循环中展示的是对齐 k*n+j 的方法。

3. cache 体系结构优化

进一步分析普通高斯消去的代码，发现：在消去部分执行 A[i][j] = A[i][j] - A[i][k] * A[k][j] 时，A[i][k] 是按列访问的，存在 cache 优化的空间，再分析这句话的复杂度是 n 的三次方，占比较大。因此，考虑以下操作进行优化：

1) 在初始时将后面会用到的 A[i][k] 转置存到 B[ ][ ] 里面；

2) 在执行 A[i][j] = A[i][j] - A[i][k] * A[k][j] 时，用 B[k][i] 代替 A[i][k] 实现行访问；

3) 把平凡算法中的 A[i][k] = 0 操作变成按行操作。

按照这样的改进思路，A[i][k]=0 由列访问到了行访问，实现了复杂度是 n 的平方（置 0）和 n 的三次方（消去）规模的 cache 优化。

为方便对比，下面列出来串行算法体系结构优化前后和并行 cache 优化的代码：

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——平凡算法

```
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;
        }}}
```

普通高斯消去——并行 + cache 优化

```
void f_pro_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++){
            float32x4_t vt=vmovq_n_f32(A[k][k]); int j;
                for (j = k + 1; j+4 <= n; j+=4){
                    va=vld1q_f32(&(A[k][j]) );
                        va= vdivq_f32(va,vt);
                        vst1q_f32(&(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=vmovq_n_f32(B[k][i]);
```

```
                        for (j = k + 1; j+4 <= n; j+=4){
                                vakj=vld1q_f32(&(A[k][j]));
                                vaij=vld1q_f32(&(A[i][j]));
                                vx=vmulq_f32(vakj,vaik);
                                vaij=vsubq_f32(vaij,vx);
                                vst1q_f32(&A[i][j], vaij);
                        }
                        for(; j<n; j++){
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
    }}}}
```

（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--) { ... }
    }
```

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]));//load Pas
        uint32x4_t vaAct = vld1q_u32(& (Act[index][k]));//load Act
        vaPas = veorq_u32(vaPas,vaAct);//向量按位异或
        vst1q_u32( &(Pas[j][k]) , vaPas );//store
    }
    for( ; k<263; k++ )//剩余部分的处理
        Pas[j][k] = Pas[j][k] ^ Act[index][k];
//*******************并行优化部分***********************
```

四、 实验结果及分析（基于鲲鹏，其他优化结果见下面章节）

（一） 普通高斯消去

1. 不同问题规模的结果

如表1所示：

问题规模	200	500	1000	2000	3000	4000
串行时间/ms	19.284	301.546	2434.87	19620.7	51208.3	158788
并行时间/ms	12.712	197.319	1592.47	12837.6	43919.8	108591
加速比	1.517	1.528	1.529	1.528	1.166	1.46

表 1: 普通高斯消去（不同问题规模）实验结果

由实验结果可知：

1) 并行比串行算法有着明显的提速，但是加速比只达到了 1.5 左右

2) 不同问题规模的加速比大致相同（怀疑 n=3000 时较小是因为访存、时间测量等原因，因为在 n=3000 前后的加速比都在 1.5 左右，没有明显的数学关系）

加速比未达到 4 的原因：

1) 访存开销：load、store 到向量的开销。特殊的，当访存地址未对齐的时候，也会影响速度。

2) 未向量化部分的开销：由于规模不一定是 4 的倍数，所以存在不能并行的部分，这部分的存在也会降低加速比。

3) 未并行部分的开销：像 A[k][k]、A[i][k] 的赋值操作等不可并行的部分也会影响整体的加速比。

4) 其他开销：如函数调用。

2. 对齐操作对实验结果的影响

如表2所示：

问题规模	200	500	1000	2000	3000
串行时间/ms	19.284	301.546	2434.87	19620.7	51208.3
并行时间/ms	12.712	197.319	1592.47	12837.6	43919.8
并行 + 对齐时间/ms	11.873	181.194	1441.07	11519.5	40220.2

表 2: 普通高斯消去（对齐操作对比）实验结果

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：

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（两部分）时间/ms	12.712	197.319	1592.47	12837.6	43919.8
并行（仅除法并行）时间/ms	19.261	302.944	2430.39	19622.4	54467.4
并行（仅消去并行）时间/ms	12.762	197.923	1604.71	12853.1	43779.8

表 3: 普通高斯消去（不同部分对比）实验结果

1) 仅对除法（二重循环）并行的时间不比串行时间短，有时甚至比串行时间还长，原因一是除法（二重循环）对代码的整体运行速度影响较小，二是并行优化存在 load、store 访存等开销，当访存等开销对结果的负影响比并行本身的正优化效果还大时，就会出现仅对除法并行的时间比串行还长的现象。这提醒我们：在并行优化时，要考虑访存本身对问题带来的影响，若为了优化而导致访存开销大于优化程度，反而得不偿失。

2) 仅对消去（三重循环）并行的时间与完全并行时间大致相等，略微偏大，说明主导普通高斯消去算法时间的是三重循环。

进一步对比分析两个循环优化（复杂度）：

除法（二重循环）中每个循环步进行了一次 load、一次向量除法运算、一次 store；总共进行了 n 的平方次循环，共 n 的平方次 load、向量除法、store。

消去（三重循环）中每个循环步进行了两次 load、两次运算、一次 store；总共进行了 n 的三次方循环。可见，三重循环的复杂度更高，执行的指令更多（比一个数量级还大），这就不难解释为什么三重循环对整体的影响占极大的地位了。

4. cache 体系结构优化的结果

在鲲鹏上的结果如表4和表5。

问题规模	200	500	1000	2000	3000
平凡算法（串行）/ms	19.288	300.876	2509.59	20975.7	58691.8
cache 优化（串行）/ms	19.357	300.75	2503.35	19635.5	52580.6

表 4: 普通高斯消去串行（体系结构优化）实验结果

可以发现，在串行算法中，当问题规模不大时，cache 优化版本并未缩短时间，而随着问题规模越来越大，cache 优化的效果越来越明显。

问题规模	1000	2000	3000
并行算法（未优化 cache）/ms	1589.59	12853.1	43765.6
并行算法（优化 cache）/ms	1606.84	12930.2	46745.4

表 5: 普通高斯消去并行（体系结构优化）实验结果

然而，在并行算法中，cache 优化版本反而更慢，这可能是由于将 A[ ][ ] 转置赋给 B[ ][ ] 的开销比消去运算的按行优化开销大，所以没有达到优化的效果。

（二） 特殊高斯消去

1. 不同问题规模的结果

使用提供的测试样例，结果如表6。
测试样例 7：矩阵列数 8399，非零消元子 6375，被消元行 4535
测试样例 8：矩阵列数 23045，非零消元子 18748，被消元行 14325
测试样例 9：矩阵列数 37960，非零消元子 29304，被消元行 14921

测试样例	7	8	9
串行时间/ms	97.412	227.282	475.283
并行时间/ms	82.391	216.58	438.493

表 6: 特殊高斯消去（不同问题规模）实验结果

可以看出并行有一定的加速效果，但由于特殊高斯算法控制流较多，不可并行的部分较多，所以加速比不如普通高斯大。

五、 探究不同指令集和平台对实验的影响

（一） 特殊高斯消去

以测试样例 8（矩阵列数 23045，非零消元子 18748，被消元行 14325）和测试样例 9（矩阵列数 37960，非零消元子 29304，被消元行 14921）问题规模为例：

1. SSE 指令集 (x86)

特殊高斯消去——SSE 并行优化部分

```
//********************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];
        va_Pas = _mm_loadu_ps((float*)&(Pas[j][k]));
        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)

特殊高斯消去——AVX256 并行优化部分

```
//*******************并行优化部分***********************
```

```
    int k;
    for (k = 0; k + 8 <= Num; k += 8){
        va_Pas2 = _mm256_loadu_ps((float*)&(Pas[j][k]));
        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)

特殊高斯消去——AVX512 并行优化部分

```
//*******************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. 结果对比

本文测试了个人笔记本（x86）和 intel devcloud 两种平台下的数据，实验结果如表7。

（个人笔记本配置：cpu 型号——Intel i5-10300H，操作系统——Windows，实验环境——visual studio2019）

测试样例	8	9
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

表 7: 特殊高斯消去（不同指令集和平台）实验结果

平台对比：在 vs2019 运行整体比 devcloud 慢。

指令集对比：在 vs2019 上，avx 的速度比 sse 更快，这主要是因为 avs256 是 8 路并行，而 sse 是 4 路并行。但是，在 Intel devcloud 上，反而并行度越高，速度越慢，根据理论分析，这应

该是由于特殊高斯消去算法的控制流更多，真正用于并行的部分占比较小，当并行路数越多时，意味着向量化的开销越大（向量更长），不可用于并行的部分也更多（mod4 和 mod8、mod16 的区别），所以可能出现并行路数增多反而效果不理想的情况。

（二） 普通高斯消去

1. SSE 指令集 (x86)

普通高斯消去——SSE 并行优化部分

```
//********************并行优化部分************************
__m128 va, vt, vx, vaij, vaik, vakj;
void f_sse(){
        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;
                for (j = k + 1; j + 4 <= n; j += 4){
                        va = _mm_loadu_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])
                            ;
                        for (j = k + 1; j + 4 <= n; j += 4){
                                vakj = _mm_loadu_ps(&(A[k][j]));
                                vaij = _mm_loadu_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[i][k] * A[k][j];
                        }
                        A[i][k] = 0;
        }}}
//********************并行优化部分************************
```

2. AVX-256 指令集 (x86)

3. AVX-512 指令集 (x86)

AVX-256 和 AVX-512 的代码和 SSE 类似，只需要把_mm_（SSE）改成_mm256_（AVX-256）和_mm512_（AVX-512）；并且把循环中的 4 改成 8 和 16。

4. 结果对比

本文测试了个人笔记本（x86）和 intel devcloud 两种平台下的数据，实验结果如表8。（个人笔记本配置：cpu 型号——Intel i5-10300H，操作系统——Windows，实验环境——visual studio2019）

测试样例	300	500	1000	2000	3000
SSE（dev）/ms	4.36	7.912	80.62	517.78	2542.37
AVX-256（dev）/ms	3.84	6.14	57.55	471.40	2317.17
AVX-512（dev）/ms	1.6	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

表 8: 普通高斯消去（不同指令集和平台）实验结果

平台对比：与特殊高斯一样，在 vs2019 运行整体比 devcloud 慢。

指令集对比：无论是 vs2019 还是 devcloud，在相同规模下，普通高斯在 sse、avx256、avx512 的速度越来越快，这是因为他们的并行度越来越高（从四路到八路再到十六路）。并且可以看出，规模越小，不同指令集运行时间的比值越接近并行路数之比。

（三） 指令集对齐

以 SSE 为例，将地址移到 16 字节对齐位置再开始并行运算，指令集对齐代码如下：

普通高斯消去——SSE 指令集对齐

```
//*******************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**********************
```

其他指令的对齐代码类似（loadu 改为 load），但是对齐的字节数不同，受文章篇幅限制，不再展示代码。

在 VS2019 的运行结果显示：在规模为 1000 的程序中，SSE 不对齐时间为 1064.99ms，对齐时间为 730.38ms，其他规模和指令集下，对齐代码均比未对齐代码运行时间短，效率高。

六、 算法探究中所用到的其他优化方法

1. 减少创建向量的指令：

只需要在程序开始时初始化所用到的向量，在程序执行过程中直接使用该向量即可。不要在程序执行过程中边定义边使用。

2. 二维数组使用连续内存，不要在堆区用 new 按行创建。

七、 profiling

（一） vtune

以规模 1000 的普通高斯消去算法为例，分析 x86 的几个指令集下的程序性能如表9

函数	Clockticks	Instructions Retired	CPI Rate	CPU Time
串行平凡算法	3,135,000,000	6,342,500,000	0.494	0.745s
sse 优化	1,850,000,000	2,860,000,000	0.647	0.440s
sse 优化 + 对齐	1,820,000,000	2,867,500,000	0.635	0.446s
avx256 优化	1,035,000,000	1,475,000,000	0.702	0.249s

表 9: vtune 分析

可以看出，指令集的并行路数越多，循环周期越少，指令条数也越少，CPU 时间越少，符合之前的实验结果。同时，发现并行度越高，CPI 越大，评价每条指令执行的周期数越大。

（二） perf

以测试样例 9 的特殊高斯消去算法为例，分析基于 arm 架构的 neon 指令集下的程序性能如表10

算法	instructions	cycles	IPC
串行算法（特殊高斯）	3,912,057,163	1,270,683,425	3.08
neon 优化（特殊高斯）	3,496,600,137	1,175,559,820	2.97

表 10: perf 分析

与上一节的普通高斯消去在 x86 指令集的 vtune 分析相似，与串行算法相比，neon 并行的循环周期更少，指令条数也更少，CPU 时间更少，符合之前的实验结果。并且，neon 优化算法的 IPC 更小，CPI 更大，与普通高斯消去趋势一致，这应该是由于 CPI 和指令集相关，而这几个指令集不同导致的。

八、 源代码

GitHub 仓库地址：https://github.com/hanmaxmax/parallel_homework2