南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

期末研究报告

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 7 月 10 日

摘要

Gröbner 基在解决密码学领域中的多变元多项式方程系统的问题中有着重要作用。而 **Gröbner 基**求解算法的优化一直是学者研究的对象。本文选取布尔 Gröbner 基计算中的一种消元子模式高斯消元算法（下文都记为“**特殊高斯消元**”），对其并行优化。

对于数据结构方面，我们采用了稠密矩阵存储，并结合数据集的特点（有限域 GF(2) 上的 0 或 1），按位存储到整型数组中，相对而言节约了存储空间，并根据消元子和被消元行的特点，分别为它们设计了不同的读取、存储方法。

对于串行算法设计，我们实现了两种算法，第一种算法按照正常的运算流程编写。但由于算法一只能整体程序的一部分——异或运算并行化，而在问题规模不大时，这种方法的优化力度不足。因此，我们设计了算法二，通过调整了部分运算的顺序，使之能够更为适合并行化，且其并行化的结果要优于算法一。

对于并行算法设计，我们用 **SIMD、Pthread、OpenMP、MPI、CUDA** 分别进行优化，并且在 MPI 编程时，结合了 SIMD 向量化和 OMP 多线程来综合实现。这几种并行优化算法的最终结果在不同体系架构上均得到了一定的优化。

我们还进行了 **Intel OneAPI** 拓展研究，分析了 DPC++ 代码编写架构，并用 DPC++ 移植特殊高斯的代码，在本地配置 DPC++ 编译环境后进行了实验及其分析。

通过对特殊高斯消元问题的研究，我们成功编写并加速了特殊高斯消元的程序，掌握了 **SIMD、Pthread、OpenMP、MPI、CUDA** 编程的思想和知识，并通过自我探究学习了 **Intel OneAPI** 的相关内容，将其应用于本文问题的优化。

关键字：SIMD、Pthread、OpenMP、MPI、CUDA、Intel OneAPI、Gröbner 基、特殊高斯消元

目录

一、 问题的提出

(一) 背景介绍

1. Gröbner 基

Gröbner 基理论的形成，经历了几十年的时间。1927 年，F.S.Macauly 为了研究理想的某些不变量，将全序的概念引入到由多变元多项式环中单项式全体组成的集合内。随后，1964 年，H.Hironaka 在研究奇性分解时，引入了多变元多项式的除法算法。1965 年，奥地利数学家 B.Buchberger 使用除法算法系统地研究了域上多变元多项式环的理想生成元问题，在单项式的集合中引入了保持单项式的乘法运算的全序（称为项序），以保证多项式相处后所得余多项式的唯一性。B.Buchberger 引入了 S-多项式，使得对多项式环中的任一给定的理想，从它的一组生成元出发，可以计算得到一组特殊的生成元，也就是 Gröbner 基。并且，Buchberger 设计了计算多元多项式理想的 Gröbner 基算法（称为 Buchberger 算法），提出了优化该算法的若干准则（Buchberger 第一、第二准则）。

Gröbner 基方法是求解非线性代数系统的一种非数值迭代的代数方法。其基本思想是，在原非线性多项式代数系统所构成的多项式环中，通过对变量和多项式的适当排序，对原系统进行约简，最后生成一个与原系统的等价且便于直接求解的标准基（Gröbner 基）。

Gröbner 基的理论和算法提供了一种标准的方法，能够很好地解决可以被表示成多变元多项式集合中的项的形式所构成的很多问题，在代数几何、交换代数和多项式理想理论、偏微分方程、编码理论、统计学、非交换代数系统理论等有广泛的应用。

2. Gröbner 基与密码学领域

Gröbner 基最主要的应用之一是辅助解决密码学领域中的多变元多项式方程系统的问题，也是多变量密码的核心问题。多变量密码的公钥就是一组有限域上多变量二次多项式。解开这组多项式方程组，就能假冒签名，读取加密信息。而 Gröbner 基就是求解多元非线性方程组的工具，Gröbner 基允许我们有效地列出其所有解的集合。事实上，计算相关系统的 Gröbner 基的工作量非常大，而基本的求解 Gröbner 基的 Buchberger 算法效率很低，因此出现了许多改进方法，目前主流的生成算法包括 F4 和 F5 算法。

然而，Gröbner 基的计算是 NP 困难的（Gröbner 基可用于求解 SAT 问题），不过，可以通过发现和利用给定系统的其他隐藏结构特性来改进计算 Gröbner 基的代数算法。在一种布尔 Gröbner 基计算 [3] 中，引入了消元子模式的特殊高斯消元方法。在 HFE80 的 Gröbner 基计算过程中，该形式的高斯消元时间占比可以达到 90% 以上。考虑到此算法中高斯消元的特殊性，为加快高斯消元速度，我们可以将此算法并行化，这也就是本研究报告所要讨论的问题。[7]

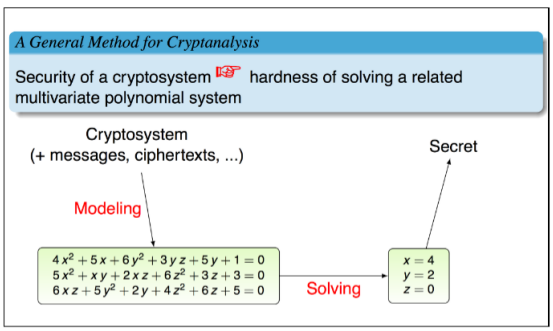


图 1: 破解密码体制到求解多项式方程组的转化

(二) 前人研究

Buchberger 提出了第一个 Gröbner 基求解算法，而后 Lazard 引入了线性代数方法，再随后 Faugère 提出了基于线性代数的 F4 和基于标签的 F5 算法。F4、F5 算法是目前公认的两个高效 Gröbner 基求解算法。F5 算法避免了 Gröbner 基的计算中的一些冗余计算，并且还特别加入了一些附加标准用以检测并减少无效计算，但整体执行效率并不高。

而在上述科学家的研究基础上，关于 Gröbner 基计算的方法以及并行算法也引起了不少人关注和研究。

例如，在基于标签的 Gröbner 基算法上有 Arri-Perry(AP) [2]、Gao-Guan-Volny(G^2V) [4] 和 Gao-Volny-Wang(GVW) 算法 [5]。在这些算法的基础上，前人还提出了对 Macaulay 矩阵的并行化改造，以及更高效的约化准则 [6]。

此外，研究者还提出了利用 MPI 技术并行处理主流算法里的中间项的约化部分，简化大型稀疏矩阵，采用并行高斯全选主元消去法等优化方法。[7]

Rodrigo Alexander Castro Campos 等人提出了布尔 Gröbner 基的有效实现，描述了计算 Gröbner 基的 F4 算法变体，包括用于检测冗余工作的 Buchberger 准则的新并行实现，以及对待定多项式的改进处理。[3]

(三) 问题介绍

1. Gröbner 基与高斯消元

线性空间中的成员判定问题是最基本的数学问题之一：任给 n 维线性空间中的向量 v_1,...,v_k 和 v, 如何判断 v 是否属于 v_1,...,v_k 所张成的线性子空间 V? 解决该问题的方法之一是将 v_1,...,v_k 和 v 作为列向量构造一个矩阵，然后用高斯消去法将矩阵化为阶梯形。那么 v 属于 V 当且仅当不存在仅有最后一个元素非零的行。

将上述问题升级到非线性空间，任给环 $K[x_1, ..., x_n]$ 中的多元多项式 f_1,...,f_k 和 f，如何判断 f 是否属于 f_1,...,f_k 生成的理想？对于多项式理想的成员判定问题，其完整解决需要用到强大的计算代数工具——Gröbner 基。

在布尔 Gröbner 基的计算中，提出了一种特殊的高斯消元方法：

2. 消元子模式的高斯消元算法

- 在这种特殊高斯消元的方法中，所基于的数据具有以下特点：

1. 运算数方面：特殊高斯消去法均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。

2. 运算规则方面：加法时——0+0=0、0+1=1、1+0=1、1+1=0；乘法时——0*0=0、0*1=0、1*0=0、1*1=1。

3. 运算矩阵：矩阵的行分为两类，“消元子”和“被消元行”，这两类行在给出矩阵的时候就已经指定。

4. “消元子”：“消元子”是在消去过程中充当“减数”的行。所有消元子的首个非零元素（即首个 1，称为首项）的位置都不同，但不会涵盖所有对角线元素。（可通过将消元子放置在特定行来令该元素位于矩阵对角线上）

5. “被消元行”：被消元行在消去过程中充当“被减数”，但在运算过程中，经过变换有可能恰好包含消元子中缺失的对角线 1 元素，此时它“升格”为消元子，补上此缺失的对角线 1 元素。

- 特殊高斯法运算的具体过程如下：

1) 逐批次将消元子和被消元行读入内存（假设实际运算的矩阵规模很大，消元子和被消元行的数量很多，可能达到百万级，大大超出内存容量）【或者】全部读入内存，不分批次（矩阵规模不大）。

2) 对当前批次中每个被消元行，检查其首项，如有对应消元子，则将其减去对应消元子，重复此过程直至其变为空行或首项不在当前批次覆盖范围内、或首项在范围内但无对应消元子或该行。

3) 运算后若某行的首项不在当前批次覆盖范围内，则该行此批次计算完成；

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；

如其首项被当前批次覆盖，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) 重复上述过程，直至所有批次都处理完毕，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）。

二、 问题分析

上一节我们给出了消元子模式的高斯消元算法（下面都记为“特殊高斯消去”）的具体过程，但是在该算法的实现中，存在以下难点。

- 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。

三、 算法设计

(一) 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。

Algorithm 1 初始化消元子 $Act[\,]$

```
ifstream infile("act.txt")
// 从文件中提取行
while infile.getline(fin, sizeof(fin)) do
    std :: stringstreamline(fin)
    flag ← 0
    // 从行中提取单个的数字
    while line >> a do
        // 取每行第一个数字为行标（二维数组第几行）
        if flag == 0 then
            index ← a
            flag ← 1
        end if
        // 按位存入 0/1
        Act[index][end − a/32]+ = 1 << (a%32)
        // 标记该消元子非空
        Act[index][end] ← 1
    end while
end while
```

Algorithm 2 初始化被消元行 $Pas[\,]$

```
ifstream infile("pas.txt")
// 从文件中提取行
while infile.getline(fin, sizeof(fin)) do
    std :: stringstreamline(fin)
    flag ← 0
    index ← 0
    // 从行中提取单个的数字
    while line >> a do
        // 用二维数组每行的最后一个位置存放被消元行每行的第一个数字
        // 即第一个被消元的位置
        if flag == 0 then
            Pas[index][end] ← a
            flag ← 1
        end if
        // 按位存入 0/1
        Pas[index][end − a/32]+ = 1 << (a%32)
    end while
    // 数组的行存储顺序与磁盘文件的存储顺序一致
    index + +
end while
```

(二) 串行算法设计

1. 串行算法一

符号说明：

$Act[\ ]$：消元子按位存储的稀疏矩阵

$Pas[\ ]$：被消元行按位存储的稀疏矩阵

$lieNum$：矩阵列数

$pasNum$：被消元行行数

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。

伪代码：

Algorithm 3 procedure LU 串行算法一

```
for i = lieNum − 1; i − 8 >= −1; i− = 8 do
    for j = 0 → pasNum do
        while i − 7 <= Pas[j][end] <= i do
            if Act[Pas[j][end]][end] == 1 then // 消元子不为空
                for k = 0 → Num do
                    Pas[j][k] = Pas[j][k]^Act[Pas[j][end]][k]
                end for
                for k = 0 → Num do
                    if Pas[j][k]! = 0 then
                        找到 Pas[j][k] 中对应的第一个非 0 位坐标 e
                        break
                    end if
                    // 更新该行的第一个非 0 坐标（存在数组该行的最后一个位置处）
                    Pas[j][end] ← e
                end for
            else// 消元子为空
                // Pas[j][ ] 来补齐消元子 Act[Pas[j][end]]
                Act[Pas[j][end]] ← Pas[j]
                // 设置消元子非空
                Act[Pas[j][end]][end] ← 1
            end if
        end while
    end for
end for
```

2. 串行算法二

在上述的串行算法设计中，我们主要是按照正常的步骤顺序，进行代码的设计。但是，在并行化这种算法时，我们发现，将向量拆分，子向量的异或自然形成任务，可分配给不同的计算单元。这是一种较为简单的并行化方式。

我们发现，异或操作是第三重子循环，其复杂度在 $O(n^3)$。当矩阵规模大到百万级别时，采取这种并行方式已经足够。但当矩阵规模没有那么大时，一个消元操作计算量不足以支撑较大规模并行，需要考虑消元操作间的并行。

但是，如果在原有算法的基础上，考虑消元操作间的并行，存在下述问题：

在串行算法一中，我们分批对消元子进行读取，然后使用当前这批消元子对所有被消元行进行消去，若被消元行没有对应的消元子则直接将其升格，作为消元行进入到后续的消元中。然而，这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即**“升格”的存在会造成程序前后的依赖关系**。因此，对特殊高斯消去的设计难点集中于两点——**如何任务划分**以及**如何处理造成依赖关系的“升格”**。

我们发现，如果不考虑升格，则消元子对不同的被消元行进行处理互不冲突，可以将被消元行划分给不同的任务，每个任务负责对一部分的被消元行进行处理，这样就可以解决**任务划分问题**。而对于另一个问题**“升格”的依赖处理**，我们考虑将升格操作单独拿出来，不参与到并行运算。该种方法对应的串行算法步骤如下：

（1）每一轮先不升格地处理被消元行：若遍历到的被消元行在消元子上能够被消元，则进行消元处理【这部分可用于后续并行任务划分】

（2）对可以升格的被消元行升格：被消元行都处理完之后，应该遍历所有新处理后的被消元行，判断其是否可以填补消元子的空缺，若能，则进行升格处理【这部分只能串行操作】

（3）若步骤（2）有被消元行被升格，则使用更新后的消元子进入下一轮的消元，即重复步骤（1）；

（4）直到步骤（2）所有被消元行都不用再升格，说明消元完成，退出程序。

分析上述算法步骤，步骤（1）中被消元行之间不存在任何依赖，可以划分为多任务；但步骤（2）遍历新被消元行时，只能串行进行，这是因为，“升格”之间存在依赖，若有两个被消元行 A B 都能填补消元子 $Act[j]$，但前一个被消元行 A 升格后，后一个被消元行 B 就不能升格了。因此，无法并行操作。

具体步骤见下面的伪代码。

(三) 并行算法设计

1. SIMD

SIMD，即 Single Instruction Multiple Data，对独立的算术运算将其循环向量化，使得每个循环步由原来的单个运算打包为向量运算，减少循环步数。

在特殊高斯的 SIMD 并行化时，我们需要找到并行化的部分，使之能够满足：

（a）有连续的内存访问；

（b）算术运算独立，没有依赖关系；

（c）有可向量化的循环。

由于算法的控制流较多，并非所有部分都能够完美使用 SIMD 并行，因此，我们优化的核心是位于最内层三重循环的异或操作。

在 arm 架构下，我们使用 neon 指令进行优化；在 x86 架构下，我们使用 SSE、AVX256、AVX512 进行优化。由于我们要计算的数据类型是 4 字节（32 位）的 int 类型，所以 neon、sse 是四路并行，AVX256 是八路并行，AVX512 是十六路并行。

Algorithm 4 procedure LU 串行算法二

```
do
for i = lieNum − 1; i − 8 >= −1; i− = 8 do
    for j = 0 → pasNum do
        while i − 7 <= Pas[j][end] <= i do
            if Act[Pas[j][end]][end] == 1 then // 消元子不为空
                for k = 0 → Num do
                    Pas[j][k] ← Pas[j][k]^Act[Pas[j][end]][k]
                end for
                for k = 0 → Num do
                    if Pas[j][k]! = 0 then
                        找到 Pas[j][k] 中对应的第一个非 0 位坐标 e
                        break
                    end if
                    // 更新该行的第一个非 0 坐标（存在数组该行的最后一个位置处）
                    Pas[j][end] ← e
                end for
            else// 消元子为空
                break
            end if
        end while
    end for
end for
sign ← false
for i = 0 → pasNum do
    temp ← Pas[j][end]
    if temp == −1 then continue
    end if
    if Act[temp][end] == 0 then
        // Pas[i][ ] 来补齐消元子 Act[temp]
        Act[temp] ← Pas[i]
        // 将被消元行升格
        Pas[i][end] ← −1
        // 标志设为 true，说明此轮还需继续
        sign ← true
    end if
end for
while (sign)
```

2. pthread

pthread 是多线程编程，采用 POSIX 标准，程序员需要控制线程管理和协调、分解并行任务并管理任务调度。

对串行算法二的 pthread 并行算法步骤：（如图2）

1) 每一轮将被消元行划分给不同线程，多线程不升格地处理被消元行；
2) 所有线程处理完之后，一个线程对可能升格的被消元行升格，其他线程等待；
3) 该线程升格完之后，通知其他线程使用更新后的消元子进入下一轮的消去；
4) 直到所有被消元行都不用再升格，说明消去完成，退出程序。

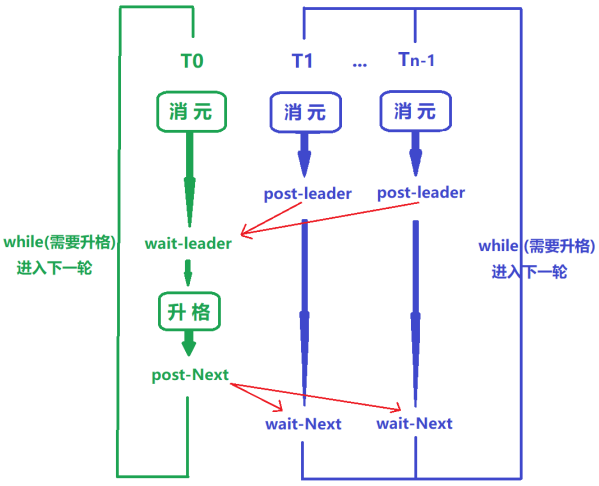


图 2: 特殊高斯 pthread 算法示意图

3. OpenMP

OpenMP 是一种可移植共享内存多线程编程规范。pthread 编程相对低层，而 OpenMP 隐藏栈管理，程序员只需将程序分为串行和并行区域，而无需构建并发执行的多线程。

对于特殊高斯消元的多线程算法，我们可以根据串行算法一和二，采用两种多线程方式：

- 方法一

只考虑对最内层循环——异或循环进行多线程划分。无需更改串行算法一的思路（即原来的串行算法不会有数据依赖被破坏），只需在异或之前划分多线程即可。

- 方法二

与 pthread 的设计类似，我们将消元部分划分给多个线程来并行执行，而升格部分串行执行。不过相比于 pthread 需要手动控制各线程何时执行、何时等待，OpenMP 只需要区分出并行和串行部分，然后在前面添加 OMP 命令即可。

4. MPI

与 pthread、OpenMP 不同，MPI 编程采用消息传递式并行程序设计，基于分布式内存体系结构，每个系统有自己独立的地址空间，处理器必须显式通信才能共享数据。而在通信方面，进程间由于地址空间独立，通过传递消息进行通信，包括同步和数据移动。因此，在设计特殊高斯消去的 MPI 并行算法时，需要基于进程间的同步和数据移动两方面实现。

首先考虑串行算法一，之前我们提到，这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即“升格”的存在会造成程序前的后依赖关系。在 MPI 程序的设计中，由于这些依赖关系的存在，不同进程需要及时知道在它负责的任务之前的进程是否有“升格”行为，若 P_k 进程的任务需要升格，则其应在 P_{k+1} 进程的任务之前进行，但前提是我们并不知道 P_k 进程什么时候需要升格，并且大部分时间内，P_k 进程是不需要升格的。因此，若每个进程都要等待它前面的进程升格与否的结果，一是通信开销巨大，二是根本没那么多“升格”操作，大部分进程都变成了无用的等待，等待开销也巨大，且浪费资源，因此，串行算法二更适合 MPI 编程的特殊高斯消去算法。

如果不考虑升格，则消元子对不同的被消元行进行处理互不冲突，可以将被消元行划分给不同的进程，每个进程负责对一部分的被消元行进行处理。因此，我们考虑将升格操作单独拿出来，不参与到并行运算。具体算法流程如下：

1) 每一轮将被消元行划分给不同进程，各进程不升格地处理自己的被消元行；

2) 所有进程处理完之后，传回数据给一个进程，使其对可能升格的被消元行升格，并设置是否还需要下一轮消去的标志 sign=true。若该轮中存在新升格的消元子，则设置标志位 sign=true，否则为 false。

3) 根据 sign 判断，若 sign 为 true，说明还需要下一轮消去。传消息并划分任务给所有进程，使用更新后的消元子进入下一轮的消去；

4) 直到所有被消元行都不用再升格，说明消去完成，退出程序。

对上述算法进行 MPI 程序的设计时，总共分为消去和升格两个部分，这两个部分的算法如下：

- **消去部分**

1) 0 号进程将被消元行划分给不同的进程，每个进程负责自己那部分的被消元行；

2) 其他进程接收任务之后直接开始运算，所有循环步遍历结束，没有自己的任务之后，传回该进程的运算结果给 0 号进程；

3) 0 号进程划分完任务后，执行自己的运算任务，此后接收来自其他进程的运算结果；

4) 在每个进程内部，依次遍历各循环步，当遍历到自己的任务时，开始消去运算；未到自己的任务时，直接跳过进入下一个循环步（与普通高斯消去不同，在消去这部分的运算中，被消元行之间完全没有冲突，可以不用等待其他进程的结果）。

- **升格部分**

在上述的一轮消去过后，应对所有被消元行进行的遍历并判断哪些行需要升格。由于这个升格过程中，后面判断到的被消元行是否升格是依赖于它前面的被消元行在该位置的升格结果，因此，升格需要串行，且只能在一个进程内完成。由于最后数据会传回 0 号进程，所以我们让 0 号进程完成这个串行操作。但是这就设计到 MPI 程序特有的**数据运输与同步**的两个关键问题：在 0 号进程完成升格后如何向其他进程传达是否需要升格（是结束循环 or 继续划分任务）的信息以及如何统一各进程数据：

1) 关于升格的选择信息，我们只需要在 0 号进程完成判断后，将标志 sign 传给其他各进程，其他进程根据标志来判断是结束循环、退出程序还是继续接收任务、执行任务。

2) 关于数据统一，我们需要在各进程执行完任务后，将其结果传回 0 号进程，而此时，该进程进入等待，待 0 号进程传回 sign 标志，再由此决定下一步的走向。这样，0 号进程就有了所有被消元行被不同进程运算后的数据，其才可以遍历所有行进行升格操作。

5. CUDA

上述并行算法，都是 CPU 应用程序，数据在 CPU 上分配，并且所有工作均在 CPU 上执行。而 GPU 编程相比于 CPU，更擅长数据并行，支持高伸缩性、高并发、足够计算精度的并行运算。在 CUDA 编程中，程序员可以控制多线程块、多线程的任务分配，以及 CPU 和 GPU 之间的数据传递。

对于特殊高斯消去的 CUDA 编程实现，我们并行化算法二的“消去”部分，划分被消元行给不同任务，将任务卸载到 GPU 上并发执行，然后统一由 CPU 进行串行的“升格”处理。

在具体实现时，由于 GPU 编程的特性，使其与 CPU 编程相比，需要重点考虑两个问题：**GPU 上内存分配**和 **host 与 device 间的数据传递与同步**。

- **内存分配**

在 CPU 编程时，我们为消元子和被消元行分别设计了二维数组 Act 和 Pas。在 CUDA 编程时，我们需要把这部分内存拷贝到 GPU。首先，需要以二维数组的大小，为 Act 和 Pas 在 GPU 开辟内存；然后在 CPU 端初始化，并将其拷贝到 GPU 上。

- **数据传递和同步**

由于具体的“消去”运算是在 GPU 端进行，而“升格”需要在 CPU 端运算，并且“升格”需要等待“消去”都完成才能开始，“升格”依赖的数据是“消去”完成后的数据。因此，我们需要在每轮大循环中：

（a）先将数据从 CPU 拷贝到 GPU

（b）GPU 并行运算

（c）GPU 运算完成后，将数据从 GPU 拷贝回 CPU

（d）CPU 串行处理“升格”，根据处理结果，决定是回到步骤（a）重复，还是结束程序。

四、 理论分析

(一) 串行算法

- **串行算法一**

设矩阵列数为 n，被消元行行数为 m。

复杂度分析：

相比于每轮处理一个消元子，我们设计的算法每轮处理 8 个消元子【第一重循环】，一定程度上拆分了循环，将最外层循环降低为原来的八分之一。在每轮内部，我们遍历所有被消元行【第二重循环】，若被消元行有对应消元子，则异或消元【第三重循环】，否则补齐消元子【第三重循环】。

其中，第一重循环的复杂度是 n/8，第二重循环是 m，第三重循环（异或消元或补齐消元子）都是对 int 型的数组操作的，并且由于数组是按位存储 0/1 的，一个 int 型数是 32 位，因此其复杂度是数组的列数，即 n/32。

因此，总的时间复杂度是 $n/8 \times m \times n/32 = mn^2/256$，即 $O(mn^2)$。

- **串行算法二**

复杂度分析：

相比于算法一，算法二是把上述的第三重循环中的补齐操作单独拿出来，作为一个独立的二重循环，并由补齐操作的结果控制是否结束程序。

在复杂度方面，算法二比算法一减少了：

（a）原算法一中，若被消元行没有对应消元子的补齐操作【第三重循环的 else 部分】

增加了：

（b）新增的统一升格环节

（c）由于单独化升格操作引起的重复消元循环

因此，算法二的复杂度是：

$times(>=2) \times \{$ { 每轮处理 8 个消元子【第一重循环】；在每轮内部，遍历所有被消元行【第二重循环】，若被消元行有对应消元子，则异或消元【第三重循环】} + 升格操作的【二重循环】}

其中 $times$ 的大小由消元子和被消元行的性质决定。

总的复杂度是 $times \times (n/8 \times m \times n/32 + m \times n/32) = times \times (mn^2/256 + mn/32)$，即 $O(mn^2)$。

(二) 并行算法

- **SIMD**

SIMD 的并行化算法，由于我们只是将异或运算向量化，但由于存在很多控制流，因此，我们预测并行化的效果不会较为接近打包的向量宽度。

- **pthread 和 OpenMP**

对于算法一的多线程设计，我们可以将主要运算——异或部分进行多线程化。但是，其一，由于串行程序过于复杂化，存在很多分支条件，导致“异或”对程序的整体性能的影响并不占绝对作用；其二，由于存放矩阵是按照每 8 位一个字节存放，而异或计算的 for 循环是遍历每个 4 字节的 int 变量来多线程并行划分，因此实际上循环划分的矩阵规模并不大（原矩阵规模的 1 / 32），所以，多线程效果可能不显著。

对于算法二的多线程并行设计中，占主要部分的运算被我们分配给多线程运行，因此，预测其加速比可能较为接近线程数。

- **MPI**

和 pthread、OpenMP 类似，MPI 的并行化也是基于算法二划分被消元行给各进程，但由于我们可以在 MPI 设计中，穿插入 OMP、SIMD 等，因此，并行化程度应较高。

- **CUDA**

相比于算法一，pthread、OMP、MPI、CUDA 针对算法二的并行的加速效果类似：

由于升格的单独化，消去的循环次数会增多，对被消元行的遍历升格次数也会增多，性能预计会下降。但是其对应的并行算法相对于较好的串行算法可能会达到优化的效果。即性能：不单独处理升格的串行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 单独处理升格的串行算法；但是，单独处理升格的并行算法 **?** 不单独处理升格的串行算法（这将在后面的实验结果给出）。

五、 实验内容与算法实现

(一) 串行算法

1. 算法一

基于伪代码，我们实现如下的串行算法：

串行算法一

```
void f_ordinary(){
    for (int i = lieNum - 1; i - 8 >= -1; i -= 8){
        //每轮处理8个消元子，范围：首项在 i-7 到 i
        for (int j = 0; j < pasNum; j++){
            //看被消元行有没有首项在此范围内的
            while (Pas[j][Num] <= i && Pas[j][Num] >= i - 7){
                int index = Pas[j][Num];
                if (Act[index][Num] == 1)//消元子不为空{
                    //Pas[j][]和Act[(Pas[j][end])][]做异或
                    for (int k = 0; k < Num; k++)
                        Pas[j][k] = Pas[j][k] ^ Act[index][k];
            //更新Pas[j][end]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 < Num; 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][Num] = S_num - 1;
                }
                else//消元子为空{
                    //Pas[j][]来补齐消元子
                    for (int k = 0; k < Num; k++)
                        Act[index][k] = Pas[j][k];
                    Act[index][Num] = 1;//设置消元子非空
                    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 (int i = lieNum % 8 - 1; i >= 0; i--) { ... }
}
```

2. 算法二

串行算法二

```
void f_ordinary(){
    bool sign;
    do{
        //不升格地处理被消元行————————————————
        //————————————————begin————————————————
        for (int i = lieNum - 1; i - 8 >= -1; i -= 8){
            //每轮处理8个消元子，范围：首项在 i-7 到 i
            for (int j = 0; j < pasNum; j++){
                //看被消元行有没有首项在此范围内的
                while (Pas[j][Num] <= i && Pas[j][Num] >= i - 7){
                    int index = Pas[j][Num];
                    if (Act[index][Num] == 1)//消元子不为空{
                        //Pas[j][]和Act[（Pas[j][x]）][]做异或
                        for (int k = 0; k < Num; k ++)
                            Pas[j][k] = Pas[j][k] ^ Act[index][k];
                    //更新Pas[j][end]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 < Num; 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][Num] = S_num - 1;
                    }
                    else//消元子为空
                        break;
                }
            }
        }
        for (int i = lieNum % 8 - 1; i >= 0; i--) { ... }
        //————————————end————————————————————
        //不升格地处理被消元行————————————————

        //升格消元子，然后判断是否结束
        sign = false;
        for (int i = 0; i < pasNum; i++){
            //找到第i个被消元行的首项
            int temp = Pas[i][Num];
            if (temp == -1)
                //说明已经被升格为消元子了
                continue;
```

```
48
49              //看这个首项对应的消元子是不是为空，若为空，则补齐
50              if (Act[temp][Num] == 0){
51                  //补齐消元子
52                  for (int k = 0; k < Num; k++)
53                      Act[temp][k] = Pas[i][k];
54                  //将被消元行升格
55                  Pas[i][Num] = -1;
56                  //标志bool设为true，说明此轮还需继续
57                  sign = true;
58              }
59          }
60      } while (sign == true);
61  }
```

(二) 并行算法

1. SIMD

在 SIMD 的优化过程中，我们将异或部分向量化，并用 neon、SSE、AVX256、AVX512 分别进行了实验，不同指令集的代码如下：

- **neon**

特殊高斯消去——neon 并行优化部分

```
1  //******************并行优化部分**********************
2      int k;
3      for (k = 0; k+4 <= Num; k+=4){
4          uint32x4_t vaPas =  vld1q_u32(& (Pas[j][k]));//load Pas
5          uint32x4_t vaAct =  vld1q_u32(& (Act[index][k]));//load Act
6          vaPas = veorq_u32(vaPas,vaAct);//向量按位异或
7          vst1q_u32( &(Pas[j][k]) , vaPas );//store
8      }
9      for( ; k<Num; k++ )//剩余部分的处理
10         Pas[j][k] = Pas[j][k] ^ Act[index][k];
11 //******************并行优化部分**********************
```

- **SSE**

特殊高斯消去——SSE 并行优化部分

```
1  //******************并行优化部分**********************
2      int k;
3      for (k = 0; k + 4 <= Num; k += 4){
4        //Pas[j][k] = Pas[j][k] ^ Act[index][k];
5          va_Pas = _mm_loadu_ps((float*)&(Pas[j][k]));
6          va_Act = _mm_loadu_ps((float*)&(Act[index][k]));
7          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**********************
```

- **AVX256**

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**********************
```

- **AVX512**

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**********************
```

2. pthread

在 pthread 并行化时，我们采用信号量的同步机制和划分被消元行的任务划分方式：

1. 定义信号量，设计线程数据结构；

2. 设计线程函数：与普通高斯消元的 pthread 并行化思路一致，为了减少线程创建、销毁的开销，以及防止代码冗余，我们将整个 do...while 循环全部纳入线程函数。

3. 串行部分的处理：由于 do...while 循环中存在串行部分，因此，我们指定一个管理其他线程的线程——0 号线程；也就是说，0 号线程需要完成“并行消元”、“串行升格”、“命令其他线程”的任务。

4. 主函数：由于我们将整个 do...while 都纳入线程函数，所以在主函数中，只需要创建线程（创建对应的 Handle + 创建对应的线程数据结构）、销毁线程、销毁信号量等即可。

5. SIMD 的结合：pthread 与 SIMD 并不冲突，在异或部分，我们同时进行 SIMD 的优化。

具体代码如下（省略与串行算法重复的代码）：

pthread 优化代码

```
int NUM_THREADS = 7;//线程数定义
//信号量定义
sem_t sem_leader;
sem_t* sem_Next = new sem_t[NUM_THREADS - 1]; // 每个线程有自己专属的信号量
struct threadParam_t
{
    int t_id; // 线程 id
};
void* threadFunc(void* param){
    threadParam_t* p = (threadParam_t*)param;
    int t_id = p->t_id;
    uint32x4_t va_Pas =  vmovq_n_u32(0);
    uint32x4_t va_Act =  vmovq_n_u32(0);
    do
    {
        for (int i = lieNum - 1; i - 8 >= -1; i -= 8)
        {
            //将被消元行分给不同线程
            for (int j = t_id; j < pasNum; j+= NUM_THREADS) {...}
        }
        for (int i = lieNum % 8 - 1; i >= 0; i--) { ... }

        //0号线程等待其它 worker 完成处理被消元行
        if (t_id == 0)
        {
            for (int i = 0; i < NUM_THREADS - 1; ++i)
                sem_wait(&sem_leader);
        }
        //其他线程完成任务后，通知0号线程；
        //然后进入睡眠，等待0号线程完成升格，再进入下一轮
        else
        {
            sem_post(&sem_leader);// 通知 leader，已完成处理被消元行
            sem_wait(&sem_Next[t_id - 1]); // 等待通知，进入下一轮
        }
        //串行：0号线程做对消元子的升格
        if (t_id == 0)
        {
            sign = false;
```

```
            for (int i = 0; i < pasNum; i++){...}
        }
        //t_id完成了升格，通知其他线程可以进入下一轮
        if (t_id == 0){
            for (int i = 0; i < NUM_THREADS - 1; ++i)
                sem_post(&sem_Next[i]); // 通知其它 worker 进入下一轮
        }
    } while (sign == true);
    pthread_exit(NULL);
}

int main(){
    init_A();
    init_P();
    //创建线程
    pthread_t* handles = new pthread_t[NUM_THREADS];//创建对应的 Handle
    threadParam_t* param = new threadParam_t[NUM_THREADS];//创建线程数据结构
    for (int t_id = 0; t_id < NUM_THREADS; t_id++)
    {
        param[t_id].t_id = t_id;
        pthread_create(&handles[t_id],NULL,threadFunc, (void*)&param[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; t_id++)
        pthread_join(handles[t_id], NULL);
    //销毁所有信号量
    sem_destroy(&sem_leader);
    sem_destroy(sem_Next);
}
```

3. OpenMP

结合 SIMD，我们对两种多线程算法的实现如下：

- **方法一：基于算法一的异或并行化**

OpenMP 优化方法一

```
void f_omp()
{
    uint32x4_t va_Pas =  vmovq_n_u32(0);
    uint32x4_t va_Act =  vmovq_n_u32(0);
    #pragma omp parallel num_threads(NUM_THREADS), private(va_Pas, va_Act)
    for (int i = lieNum-1; i - 8 >= -1; i -= 8)
    {
        for (int j = 0; j < pasNum; j++)
        {
            while (Pas[j][Num] <= i && Pas[j][Num] >= i - 7)
            {
                int index = Pas[j][Num];
```

```
                if (Act[index][Num] == 1)
                {
                    #pragma omp for schedule(static)
                    for (int k = 0; k <= Num-4; k+=4)
                    {
                        va_Pas =  vld1q_u32(& (Pas[j][k]));
                        va_Act =  vld1q_u32(& (Act[index][k]));
                        va_Pas = veorq_u32(va_Pas,va_Act);
                        vst1q_u32( &(Pas[j][k]) , va_Pas );
                    }

                    for(int k=Num-Num%4 ; k<Num; k++ )
                        Pas[j][k] = Pas[j][k] ^ Act[index][k];

                    int num = 0, S_num = 0;
                    for (num = 0; num < Num; num++)
                    {...}
                    Pas[j][Num] = S_num - 1;
                }
                else{...}
            }
        }
    }
    for (int i = lieNum%8-1; i >= 0; i--){...}
}
```

- **方法二：基于算法二的划分被消元行为多任务**

OpenMP 优化方法二

```
void f_omp()
{
    uint32x4_t va_Pas =  vmovq_n_u32(0);
    uint32x4_t va_Act =  vmovq_n_u32(0);
    bool sign;
    #pragma omp parallel num_threads(NUM_THREADS), private(va_Pas, va_Act)
    do
    {
        for (int i = lieNum - 1; i - 8 >= -1; i -= 8)
        {
            #pragma omp for schedule(static)
            for (int j = 0; j < pasNum; j++)
            {...}
        }
        for (int i = lieNum%8-1; i >= 0; i--)
        {...}

    //升格消元子，然后判断是否结束
```

```
    #pragma omp single
    {
        sign = false;
        for (int i = 0; i < pasNum; i++)
        {...}
    }
    }while (sign == true);
}
```

4. MPI

在 MPI 编程时，我们结合前述的 SIMD、OpenMP，以循环划分的方式进行实现。(具体思路见注释):

MPI+OpenMP+SIMD（SSE）优化

```
void super(int rank, int num_proc)
{
    //不升格地处理被消元行begin————————
    int i;
    //每轮处理8个消元子, 范围: 首项在 i-7 到 i
#pragma omp parallel num_threads(thread_count)
    for (i = lieNum - 1; i - 8 >= -1; i -= 8)
    {
#pragma omp for schedule(dynamic,20)
        for (int j = 0; j < pasNum; j++)
        {
            //当前行是自己进程的任务——进行消去
            if (int(j % num_proc) == rank){
                while (Pas[j][Num] <= i && Pas[j][Num] >= i - 7){
                    int index = Pas[j][Num];
                    if (Act[index][Num] == 1){//消元子不为空
                        int k;
                        __m128 va_Pas, va_Act;
                        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];
                        }

                        //更新首项值
                        int num = 0, S_num = 0;
                        for (num = 0; num < Num; 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][Num] = S_num - 1;
                    }
                    else//消元子为空
                        break;
    }}}}
#pragma omp parallel num_threads(thread_count)
    for (int i = lieNum % 8 - 1; i >= 0; i--){
        //每轮处理1个消元子，范围：首项等于i
        ......//代码与前面相似，故省略
        }
    }
    //不升格地处理被消元行end----------
}

void f_mpi()
{
    int num_proc;//进程数
    int rank;//识别调用进程的rank，值从0~size-1
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    //0号进程--任务划分
    if (rank == 0){
        int sign;
        do{
            //任务划分
            for (int i = 0; i < pasNum; i++){
                int flag = i % num_proc;
                if (flag == rank)
                    continue;
                else
                    MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, flag, 0,
                        MPI_COMM_WORLD);
            }
            super(rank, num_proc);
            //处理完0号进程自己的任务后需接收其他进程处理之后的结果
            for (int i = 0; i < pasNum; i++){
                int flag = i % num_proc;
                if (flag == rank)
```

```
                continue;
            else
                MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, flag, 1,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        //升格消元子，然后判断是否结束
        sign = 0;
        for (int i = 0; i < pasNum; i++){
            //找到第i个被消元行的首项
            int temp = Pas[i][Num];
            if (temp == -1){
                //说明他已经被升格为消元子了
                continue;
            }
            //看这个首项对应的消元子是不是为空，若为空，则补齐
            if (Act[temp][Num] == 0){
                //补齐消元子
                for (int k = 0; k < Num; k++)
                    Act[temp][k] = Pas[i][k];
                //将被消元行升格
                Pas[i][Num] = -1;
                //标志设为true，说明此轮还需继续
                sign = 1;
            }
        }
        for (int r = 1; r < num_proc; r++){
            MPI_Send(&sign, 1, MPI_INT, r, 2, MPI_COMM_WORLD);
        }
    } while (sign == 1);
}

//其他进程
else{
    int sign;
    do{
        //非0号进程先接收任务
        for (int i = rank; i < pasNum; i += num_proc){
            MPI_Recv(&Pas[i], Num + 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
        //执行任务
        super(rank, num_proc);
        //非0号进程完成任务之后，将结果传回到0号进程
        for (int i = rank; i < pasNum; i += num_proc){
            MPI_Send(&Pas[i], Num + 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
        }
        MPI_Recv(&sign, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
```

```
126
127         } while (sign == 1);
128     }
129 }
```

5. CUDA

数据结构： CUDA 编程与 CPU 编程的数据结构不同，我们需要先把二维数组“铺平”为一维数组，然后为 GPU 分配展开后的一维数组的空间，并进行 host 端、device 端的数据传递。

核函数：将原来 do...while 循环中的外层 for 循环运算卸载给核函数,以 $g_index = blockIdx.x \times blockDim.x + threadIdx.x$; 为起始位置，$gridStride = gridDim.x \times blockDim.x$ 为跨度划分被消元行。

具体代码如下:

CUDA 优化代码

```
1  __global__ void work(int g_Num, int g_pasNum, int g_lieNum, int* g_Act, int*
       g_Pas)
2  {
3      int g_index = blockIdx.x * blockDim.x + threadIdx.x;
4      int gridStride = gridDim.x * blockDim.x;
5
6      for (int i = g_lieNum - 1; i - 8 >= -1; i -= 8)
7      {
8          //给核函数分配任务——循环划分被消元行
9          for (int j = g_index; j < g_pasNum; j+=gridStride)
10         {
11             while (g_Pas[j * (g_Num + 1) + g_Num] <= i && g_Pas[j * (Num + 1)
                   + g_Num] >= i - 7)
12             {
13                 int index = g_Pas[j * (Num + 1) + g_Num];
14                 if (g_Act[index * (Num + 1) + g_Num] == 1)//消元子不为空
15                 {
16                     //Pas[j][]和Act[(Pas[j][x])][]做异或
17                     //***************SIMD优化部分*****************
18                     for (int k = 0; k < g_Num; k ++)
19                     {
20                         g_Pas[j * (Num + 1) + k] = g_Pas[j * (Num + 1) + k] ^
                               g_Act[index * (Num + 1) + k];
21                     }
22                     //***************SIMD优化部分*****************
23
24                     int num = 0, S_num = 0;
25                     for (num = 0; num < g_Num; num++)
26                     {
27                         if (g_Pas[j * (Num + 1) + num] != 0)
28                         {
29                             unsigned int temp = g_Pas[j * (Num + 1) + num];
30                             while (temp != 0)
```

```
                                {
                                    temp = temp >> 1;
                                    S_num++;
                                }
                                S_num += num * 32;
                                break;
                            }
                        }
                        g_Pas[j * (Num + 1) + g_Num] = S_num - 1;
                    }
                    else//消元子为空
                    {
                        break;
                    }
                }
            }
        }
        for (int i = g_lieNum % 8 - 1; i >= 0; i--)
        {...}
}

int main()
{
    cudaError_t ret;
    init_A();
    init_P();
    int* g_Act, *g_Pas;

    ret=cudaMalloc(&g_Act, lieNum * (Num + 1) * sizeof(int));
    ret=cudaMalloc(&g_Pas, lieNum * (Num + 1) * sizeof(int));
    if (ret != cudaSuccess) {
        printf("cudaMalloc gpudata failed!\n");
    }
    size_t threads_per_block = 256;
    size_t number_of_blocks = 32;

    cudaEvent_t start, stop;//计时器
    float etime = 0.0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);//开始计时

    bool sign;
    do
    {
        ret = cudaMemcpy(g_Act, Act, sizeof(int) * lieNum * (Num + 1),
            cudaMemcpyHostToDevice);
        ret = cudaMemcpy(g_Pas, Pas, sizeof(int) * lieNum * (Num + 1),
```

```
            cudaMemcpyHostToDevice);
78          if (ret != cudaSuccess) {
79              printf("cudaMemcpyHostToDevice failed!\n");
80          }
81
82          //不升格地处理被消元行————————————
83          work <<< 1024, 10 >>> (Num, pasNum, lieNum, g_Act, g_Pas);
84          cudaDeviceSynchronize();
85          //不升格地处理被消元行————————————
86
87          ret = cudaMemcpy(Act, g_Act, sizeof(int) * lieNum * (Num + 1),
                cudaMemcpyDeviceToHost);
88          ret = cudaMemcpy(Pas, g_Pas, sizeof(int) * lieNum * (Num + 1),
                cudaMemcpyDeviceToHost);
89          if (ret != cudaSuccess) {
90              printf("cudaMemcpyDeviceToHost failed!\n");
91          }
92
93          //升格消元子, 然后判断是否结束
94          sign = false;
95          for (int i = 0; i < pasNum; i++)
96          {
97              //找到第i个被消元行的首项
98              int temp = Pas[i* (Num + 1) + Num];
99              if (temp == -1)//说明他已经被升格为消元子了
100                 continue;
101             //看这个首项对应的消元子是不是为空, 若为空, 则补齐
102             if (Act[temp * (Num + 1) + Num] == 0)
103             {
104                 //补齐消元子
105                 for (int k = 0; k < Num; k++)
106                     Act[temp * (Num + 1) + k] = Pas[i * (Num + 1) + k];
107                 //将被消元行升格
108                 Pas[i * (Num + 1) + Num] = -1;
109                 //标志bool设为true, 说明此轮还需继续
110                 sign = true;
111             }
112         }
113     } while (sign == true);
114     cudaEventRecord(stop, 0);
115     cudaEventSynchronize(stop);//停止计时
116     cudaEventElapsedTime(&etime, start, stop);
117     printf("GPU_LU:%f ms\n", etime);
118 }
```

六、 实验结果及分析

(一) 实验环境与实验数据

我们在不同平台上进行了不同架构、不同操作系统的实验, 实验中所用到的各平台属性如下:

- **鲲鹏服务器：arm+linux**（如图3）

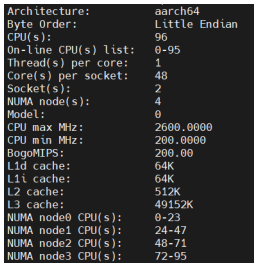


图 3: 鲲鹏服务器

- **笔记本电脑：x86+Windows**（如图4）**显卡：NVIDIA GeForce GTX 1650 Ti**

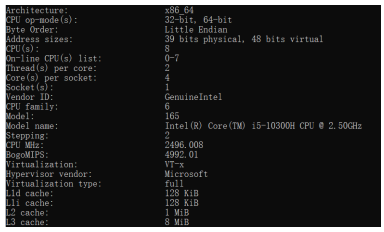


图 4: PC

- **Intel Devcloud：x86+linux**
- **金山云服务器：x86+linux**
- **使用的测试样例：**

由于 10 个测试样例中，前几个样例的运行时间不到 10ms，时间过小会导致测量误差过大，程序性能测试失真。因此，我们主要采用后面几组较大的测试样例。

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 10：矩阵列数 43577，非零消元子 39477，被消元行 54274

(二) SIMD

在不同指令集和架构（鲲鹏服务器、Intel Devcloud、笔记本电脑）进行实验的结果如表1。

优化效果：可以看到，在不同平台下，SIMD 均有一定的优化效果。但是与《理论分析》一节所预测的结果一致，由于存在较多的控制流，最后的优化加速比并未达到向量宽度。

平台对比：整体而言，devcloud 最快，其次是鲲鹏服务器，最慢的是本机上的 vs2019。

测试样例	7	8	9
串行算法（鲲鹏）/ms	97.412	227.282	475.283
neon（鲲鹏）/ms	82.391	216.58	438.493
串行算法（Devcloud）/ms	67.1	203.19	417.69
SSE（Devcloud）/ms	39.2	120.05	267.24
AVX-256（Devcloud）/ms	40.3	125.57	263.34
AVX-512（Devcloud）/ms	41.5	136.08	309.119
串行算法（vs2019）/ms	141.6	417.27	1002.5
SSE（vs2019）/ms	130.4	372.54	981.81
AVX-256（vs2019）/ms	110.61	322.27	771.66

表 1: SIMD 实验结果

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 和鲲鹏服务器上，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。

(三) Pthread 和 OpenMP

在工作线程个数为 7、测试环境为鲲鹏服务器并且结合 neon 编程的条件下，我们对以下样例进行实验，结果如表2。（基于算法一的多线程是仅并行化“异或”，基于算法二的多线程是划分被消元行）

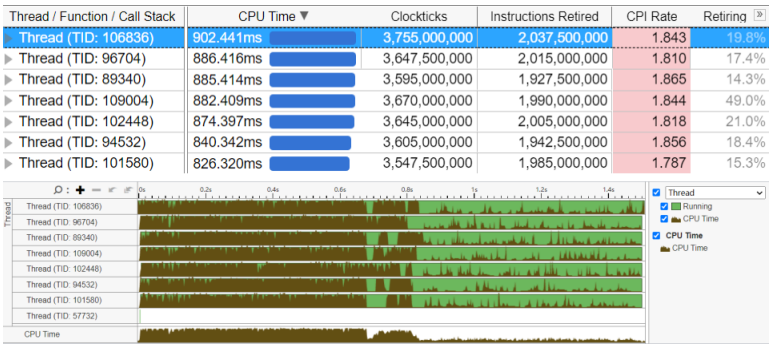
测试样例	8	9	10
串行算法一/ms	227.28	475.28	1984.17
串行算法二/ms	627.93	1158.85	4890.63
基于算法二的 pthread 算法/ms	92.98	189.26	1521.16
基于算法一的 OpenMP 算法/ms	210.8	448.12	1897.35
基于算法二的 OpenMP 算法/ms	98.58	184.75	1101.74

表 2: pthread 和 OpenMP 实验结果

从表2中可看出，预期结果与我们理论分析的结果大致相同，单独处理升格的串行算法要比不单独处理升格的串行算法慢，但是这种“较差”的串行算法要比“较好”的串行算法更适合应用于多线程编程，且其多线程处理过的程序相比于“较好”的串行算法要更快。相对于为了多线程而设计的“较差”串行程序，OpenMP 版本可以达到超过 6、接近线程数 7 的加速比；而相对于“较好的”串行程序，OpenMP 可以达到超过 2 的加速比。并且，OpenMP 在问题规模较小的时候与 Pthread 相差很小，在大规模下，要比 Pthread 更优，这与之前的普通高斯消去结果是一致的。

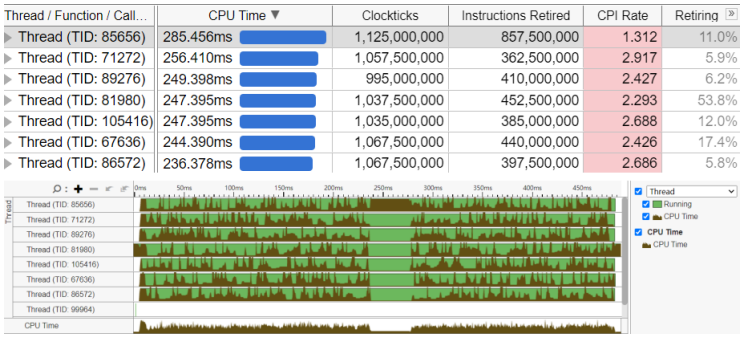
而对于只给“异或”多线程化的算法而言，其加速效果并不大，说明此种算法并不能充分利用多线程的优势。

在程序使用 7 个线程的条件下，我们用 vtune 工具对基于算法一的 OpenMP 算法和基于算法二的 OpenMP 算法进行 profiling。其中算法一的各线程剖析结果如图5，算法二的结果如图6。



Thread / Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
Thread (TID: 106836)	902.441ms	3,755,000,000	2,037,500,000	1.843	19.8%
Thread (TID: 96704)	886.416ms	3,647,500,000	2,015,000,000	1.810	17.4%
Thread (TID: 89340)	885.414ms	3,595,000,000	1,927,500,000	1.865	14.3%
Thread (TID: 109004)	882.409ms	3,670,000,000	1,990,000,000	1.844	49.0%
Thread (TID: 102448)	874.397ms	3,645,000,000	2,005,000,000	1.818	21.0%
Thread (TID: 94532)	840.342ms	3,605,000,000	1,942,500,000	1.856	18.4%
Thread (TID: 101580)	826.320ms	3,547,500,000	1,985,000,000	1.787	15.3%

图 5: 基于算法一的多线程分析



Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
Thread (TID: 85656)	285.456ms	1,125,000,000	857,500,000	1.312	11.0%
Thread (TID: 71272)	256.410ms	1,057,500,000	362,500,000	2.917	5.9%
Thread (TID: 89276)	249.398ms	995,000,000	410,000,000	2.427	6.2%
Thread (TID: 81980)	247.395ms	1,037,500,000	452,500,000	2.293	53.8%
Thread (TID: 105416)	247.395ms	1,035,000,000	385,000,000	2.688	12.0%
Thread (TID: 67636)	244.390ms	1,067,500,000	440,000,000	2.426	17.4%
Thread (TID: 86572)	236.378ms	1,067,500,000	397,500,000	2.686	5.8%

图 6: 基于算法二的多线程分析

从分析结果中可看出，算法一里多线程运行（CPU Time）主要集中在程序前半部分（CPU 时间更集中），而算法二中各线程 CPU Time 在前和后都存在，且较为分散。从代码角度分析原因可知，算法一的多线程只集中于“异或”，而算法二的多线程是划分被消元行，因此算法二全局都存在多线程的并行。

此外，我们还注意到图5中 TID 等于 85656 的线程用时要比其他线程都长，而且在中间一段的 CPU Time 中，只有 85656 的线程在工作（且工作强度较大，CPU Time 很密集）其他线程在此时都处于闲置状态。因此我们可以推测，85656 的线程是“主线程”，即我们在程序中设计，用于串行部分和控制依赖关系的线程。

抛开主线程，我们发现，其余工作线程的工作时间都较为“对齐”，说明负载较为均衡。

(四) MPI+SIMD+OMP

在进程数分别为 4、测试环境为金山云并且结合 SIMD 和 OpenMP 编程的条件下，我们对以下样例进行实验，结果如表3。

此外，我们对比了两种串行算法的 profiling 结果，串行算法一的结果如图7，算法二的结果如图8。仅就串行程序而言，算法一的速度明显快于算法二，Clockticks、Instructions Retired 和 CPI 均表现为算法一更小。这主要是由于为了适应于并行化，我们在算法二中添加了部分冗余内容，使得其整体性能都较算法一有所降低。

测试样例	8	9	10
串行算法一/ms	257.48	485.28	2084.17
串行算法二/ms	419.47	1006.49	4220.47
基于算法二的 MPI 算法/ms	204.86	418.64	1788.68

表 3: MPI 实验结果

图 7: 串行算法一的 vtune 分析

图 8: 串行算法二的 vtune 分析

但是，表格中的实验结果与 OMP、pthread 的结果相似，虽然串行算法二要比不单独处理升格的串行算法一慢，不过这种“较差”的串行算法要比“较好”的串行算法更适合应用于并行编程，且其并行处理过的程序相比于“较好”的串行算法要更快。

(五) CUDA

根据笔记本电脑的显卡型号,在本地配置 CUDA 环境,设置 $threads_per_block$ 为 256,$number_of_blocks$ 为 32，进行实验，结果如表4。

测试样例	7	8	9
串行算法一/ms	23.54	297.85	807.32
串行算法二/ms	41.41	484.92	1741.74
基于算法二的 CUDA 算法/ms	14.19	140.56	398.83
加速比	2.92	3.45	4.37

表 4: CUDA 实验结果

从结果中，可以发现，随着测试样例规模的增大，CUDA 的加速效果更好。

CUDA 程序的性能主要受到两方面的对抗制约：**并行部分的正向加速**和 **CPU、GPU 之间数据传递的反向减速**。当样例规模不大时，并行运算部分的规模小，而 CUDA 的信息传递开销过大，因此加速效果不明显。而当规模逐渐增大，并行部分的规模变大，其加速的效果超越了数据传递的开销，使得性能更好。

七、 拓展研究：Intel OneAPI

(一) 工具介绍

OneAPI 是 Intel 推出的跨架构（多核 CPU、众核 GPU、FPGA、…）统一编程模型，旨在简化多种架构下并行程序的开发流程。OneAPI 的主要组件是跨架构并行编程语言 DPC++（Data parallel C++），它是英特尔现在正在开发的一种新的语言，它是开放的、基于标准的、高性能的，并且，能够跨不同的硬件架构提供高性能。总而言之，Inter 提供了完整的开发生态，包括核心工具套件和库，可以实现支持跨架构（CPU、GPU、FPGA）开发高性能应用。[1]

传统的并行程序是在不同并行架构下（SIMD、多核 CPU、GPU），用专用编程工具/语言（SSE/AVX、Pthread、OpenMP、CUDA）对求解问题进行并行求解。而 Inter OneAPI 的 DPC++ 编程提供了能够实现代码在不同架构间的复用的形式，DPC++ 编译器可以实现针对不同架构的目标代码调优。并且，DPC++ Compatibility tool 可帮助迁移用 CUDA 编写的现有代码到 DPC++ 程序，实现代码迁移。此外，OneAPI 还提供了程序的分析调试工具。

(二) DPC++ 语法介绍

编写 DPC++ 程序，首先需要包含头文件 *#include <CL/sycl.hpp>*

DPC++ 程序在主机调用，将计算卸到 GPU。程序员需要使用 **queue, buffer, device, and kernel abstractions** 来抽象指导卸掉哪些部分的计算和数据。

具体步骤：

- **Step 1：创建一个 queue**

我们通过将任务提交到 **queue** 来将计算卸载到设备上。

程序员可以通过选择器选择 CPU、GPU、FPGA 和其他设备。

不同硬件的迁移：在创建 queue 的时候，可以指定硬件，如下面的代码所示，创建设备选择类，并在定义 queue 的时候，指定硬件“Intel”、“AMD”、“Nvidia”。如果不指定，则使用默认硬件。如图9

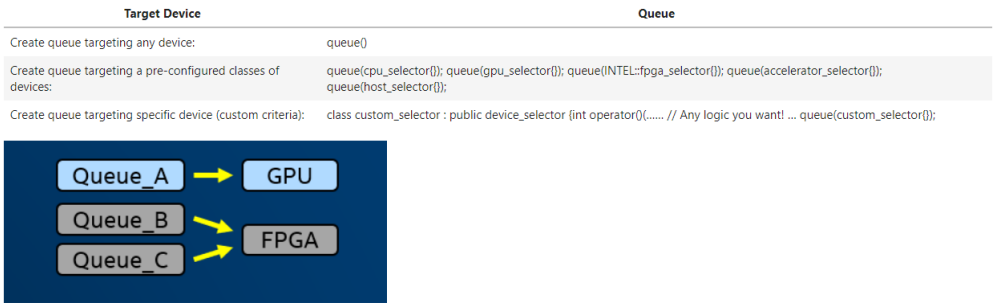


图 9: queue

```
class my_device_selector : public device_selector {
public:
    my_device_selector(std::string vendorName) : vendorName (vendorName){};
    int operator()(const device& dev) const override {
    int rating = 0;
//We are querying for the custom device specific to a Vendor and if it is a
    GPU device we are giving the highest rating as 3. The second preference
    is given to any GPU device and the third preference is given to CPU
    device.
    if (dev.is_gpu() & (dev.get_info<info::device::name>().     find(
        vendorName_) != std::string::npos))
        rating = 3;
    else if (dev.is_gpu()) rating = 2;
    else if (dev.is_cpu()) rating = 1;
    return rating;
    };
private:
    std::string vendorName_;
};

int main() {
    //pass in the name of the vendor for which the device you want to query
    std::string vendor_name = "Intel";
    //std::string vendor_name = "AMD";
    //std::string vendor_name = "Nvidia";
    my_device_selector selector(vendor_name);
    queue q(selector);
}
```

- **Step 2：创建 buffer，代表 both host and device memory**

accessor 在 SYCL 中创建数据依赖项，用于排序内核执行。如果两个内核使用相同的缓冲区，则第二个内核需要等待第一个内核完成，以避免竞争。如图10

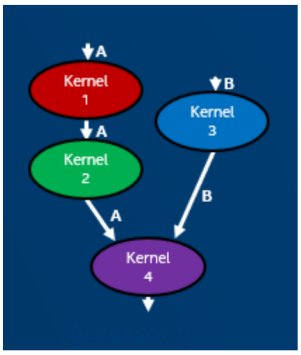


图 10

- **Step 3：提交用于异步执行的指令（q.submit）**

- **Step 4：创建缓冲区访问器 buffer accessor 以访问设备上的缓冲区数据**

设备和主机可以共享物理内存，也可以具有不同的内存。当内存不同时，卸载计算需要在主机和设备之间复制数据。DPC++ 不要求程序员管理数据副本。通过创建 **Buffers and Accessors**，DPC++ 确保数据可用于主机和设备，而无需任何程序员的努力。DPC++ 还允许程序员在需要达到最佳性能时显式控制数据移动。

- **Step 5：lamdba 函数发送 kernel 以执行**

在 DPC++ 程序中，我们定义一个内核，该内核应用于索引空间中的每个点。对于像这样的简单程序，索引空间直接映射到数组的元素。内核封装在 C++lambda 函数中。lambda 函数作为坐标数组在索引空间中传递一个点。对于这个简单的程序，索引空间坐标与数组索引相同。下面程序中的 **parallel_for** 将 lambda 应用于索引空间。索引空间在 **parallel_for** 的第一个参数中定义为从 0 到 N-1 的一维范围。

- **Step 6：写 kernel**

kernel 的调用是并行执行的。程序在执行时，会为范围的每个元素调用 kernel，而 kernel 调用可以访问调用 id。

如下示例：

```
void dpcpp_code(int* a, int* b, int* c, int N) {
  //Step 1: create a device queue
  //(developer can specify a device type via device selector or use default
      selector)
  auto R = range<1>(N);
  queue q;
  //Step 2: create buffers (represent both host and device memory)
  buffer buf_a(a, R);
  buffer buf_b(b, R);
  buffer buf_c(c, R);
  //Step 3: submit a command for (asynchronous) execution
  q.submit([&](handler &h){
  //Step 4: create buffer accessors to access buffer data on the device
  accessor A(buf_a,h,read_only);
  accessor B(buf_b,h,read_only);
  accessor C(buf_c,h,write_only);

  //Step 5: send a kernel (lambda) for execution
  h.parallel_for(range<1>(N), [=](auto i){
    //Step 6: write a kernel
    //Kernel invocations are executed in parallel
    //Kernel is invoked for each element of the range
    //Kernel invocation has access to the invocation id
    C[i] = A[i] + B[i];
    });
  });
}
```

- **其他编程范式：**

除此之外，DPC++ 还有 Unified Shared Memory（USM）、Sub-Groups 等架构。使用 USM，开发人员可以在主机和设备代码中引用相同的内存对象，如图11。而 USM 体系的编程范式与 **buffer** 不同，它的编写方式如图12。

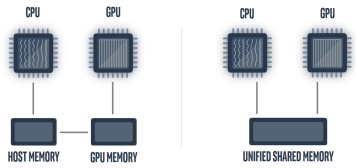


图 11: queue

Type	function call	Description	Accessible on Host	Accessible on Device
Device	malloc_device	Allocation on device (explicit)	NO	YES
Host	malloc_host	Allocation on host (implicit)	YES	YES
Shared	malloc_shared	Allocation can migrate between host and device (implicit)	YES	YES

图 12: queue

(三) 特殊高斯代码 OneAPI 移植

首先，在本地下载 ONeAPI 的 toolkit，配置 DPC++ 编译环境。然后，将特殊高斯的 CUDA 代码，通过 DPC++ Compatibility tool 移植。

移植后的代码如下（省略与之前重复的初始化等部分，完整代码在文末 GitHub 链接中查看）：

DPC++ 移植代码

```
void work(int g_Num, int g_pasNum, int g_lieNum, int *g_Act, int *g_Pas,
          sycl::nd_item<3> item_ct1)
{
    int g_index = item_ct1.get_group(2) * item_ct1.get_local_range(2) +
                  item_ct1.get_local_id(2);
    int gridStride = item_ct1.get_group_range(2) * item_ct1.get_local_range
        (2);

    for (int i = g_lieNum - 1; i - 8 >= -1; i -= 8)
    {
        for (int j = g_index; j < g_pasNum; j+=gridStride)
        {...}
    }
    for (int i = g_lieNum % 8 - 1; i >= 0; i--){...}
}

int main() try {
    dpct::device_ext &dev_ct1 = dpct::get_current_device();
    sycl::queue &q_ct1 = dev_ct1.default_queue();

```

```
    init_A();
    init_P();

    int* g_Act, *g_Pas;
    g_Act = sycl::malloc_device<int>(lieNum * (Num + 1), q_ct1);
    g_Pas = sycl::malloc_device<int>(lieNum * (Num + 1), q_ct1);

    size_t threads_per_block = 256;
    size_t number_of_blocks = 32;

    sycl::event start, stop;
    std::chrono::time_point<std::chrono::steady_clock> start_ct1;
    std::chrono::time_point<std::chrono::steady_clock> stop_ct1; //计时器
    float etime = 0.0;

    start_ct1 = std::chrono::steady_clock::now();
    start = q_ct1.ext_oneapi_submit_barrier(); //开始计时

    bool sign;
    do
    {
        q_ct1.memcpy(g_Act, Act, sizeof(int) * lieNum * (Num + 1)).wait();
        q_ct1.memcpy(g_Pas, Pas, sizeof(int) * lieNum * (Num + 1)).wait();

        //不升格地处理被消元行
        q_ct1.submit([&](sycl::handler &cgh) {
            auto Num_ct0 = Num;
            auto pasNum_ct1 = pasNum;
            auto lieNum_ct2 = lieNum;

            cgh.parallel_for(sycl::nd_range<3>(sycl::range<3>(1, 1, 256) *
                                                   sycl::range<3>(1, 1, 32),
                                               sycl::range<3>(1, 1, 32)),
                             [=](sycl::nd_item<3> item_ct1) {
                                 work(Num_ct0, pasNum_ct1, lieNum_ct2, g_Act,
                                      g_Pas, item_ct1);
                             });
        });
        dev_ct1.queues_wait_and_throw();

        q_ct1.memcpy(Act, g_Act, sizeof(int) * lieNum * (Num + 1)).wait();
        q_ct1.memcpy(Pas, g_Pas, sizeof(int) * lieNum * (Num + 1)).wait();

        //升格消元子，然后判断是否结束
        sign = false;
        for (int i = 0; i < pasNum; i++)
        {
            //找到第i个被消元行的首项
```

```
68              int temp = Pas[i* (Num + 1) + Num];
69              if (temp == -1)
70                  continue;//说明他已经被升格为消元子了
71              //看这个首项对应的消元子是不是为空，若为空，则补齐
72              if (Act[temp * (Num + 1) + Num] == 0)
73              {
74                  //补齐消元子
75                  for (int k = 0; k < Num; k++)
76                      Act[temp * (Num + 1) + k] = Pas[i * (Num + 1) + k];
77                  Pas[i * (Num + 1) + Num] = -1; //将被消元行升格
78                  sign = true;//标志bool设为true，说明此轮还需继续
79              }
80          }
81      } while (sign == true);
82
83      dpct::get_current_device().queues_wait_and_throw();
84      stop_ct1 = std::chrono::steady_clock::now();
85      stop = q_ct1.ext_oneapi_submit_barrier(); //停止计时
86      etime = std::chrono::duration<float, std::milli>(stop_ct1 - start_ct1).
            count();
87      printf("GPU_LU:%f ms\n", etime);
88  }
89  catch (sycl::exception const &exc) {
90    std::cerr << exc.what() << "Exception caught at file:" << __FILE__
91              << ", line:" << __LINE__ << std::endl;
92    std::exit(1);
93  }
```

(四) 实验结果

将 CUDA 移植后的 DPC++ 代码在本地（笔记本电脑）进行测试，结果如表5。

（**Device:**）Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 集显

测试样例	7	8	9
串行算法一/ms	23.54	297.85	807.32
串行算法二/ms	41.41	484.92	1741.74
基于算法二的 CUDA 算法/ms	14.19	140.56	398.83
由 CUDA 算法移植的 DPC++ 算法/ms	26.86	383.82	663.09

表 5: OneAPI 实验结果

从实验结果发现，在各测试样例下，由 CUDA 移植后的 DPC++ 程序都要比原 CUDA 用时长，并且 DPC++ 程序要比 CUDA 程序慢一倍。

分析其原因，我们发现，DPC++ 程序使用的设备是 Intel 集显，而 CUDA 所用的是笔记本电脑上的 NVIDIA 独立显卡。因此其性能会有一定差异。但由于实验设备有限（只有笔记本电脑），我们暂时无法在其他设备上进行程序测试。

不过我们可以肯定的是，串行算法一、二和 DPC++ 程序是在相同的主板上运行的，并且 DPC++ 的结果相比于串行算法，有加速效果。随着问题规模的增大，加速效果越明显。这也和上述 CUDA 编程的实验结果趋势一致，进一步说明了 DPC++ 有移植的效果。

八、 总结

首先，本文从 Gröbner 基问题入手，了解了 Gröbner 基的相关知识和前人对其的优化研究。然后从基于 Gröbner 基的高斯消元方法入手，进行串行算法设计和并行化研究。在串行算法设计时，我们先按照正常的运算流程编写了算法一。但由于算法一只能对部分运算——异或运算并行化，而这样的并行化在问题规模不大时，优化力度不足。因此，我们调整了部分运算的顺序，设计了算法二。虽然算法二的效果不如算法一，但算法二更适合并行化，且其并行化的结果要优于算法一。

在并行化的部分，我们可归类为 CPU 和 GPU 优化两种。在 CPU 上，我们分别用 SIMD、Pthread、OpenMP、MPI 的方法进行优化，并在 MPI 编程时，与 SIMD、OMP 相综合。在 GPU 上，我们使用 CUDA 编程。CPU 和 GPU 的几种并行优化结果均较为显著，基本可以达到 2 以上的加速比，有些优化算法还可以达到 6 以上的加速比。并且在不同平台的效果不同，一般表现为 Intel Devcloud、鲲鹏最快，金山云次之，本地较慢。

此外,我们还用 Intel OneAPI 进行了拓展实验。分析了 DPC++ 代码编写架构,并用 DPC++ 移植特殊高斯的代码，在本地配置 DPC++ 编译环境后进行了实验及其分析。

总之，通过本文的研究内容，我们成功编写并加速了特殊高斯消元的程序，掌握了 SIMD、Pthread、OpenMP、MPI、CUDA 编程的思想和知识，并通过自我探究学习了 OneAPI 的相关内容，将其应用于本文问题的优化。

九、 源代码

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

参考文献

[1] Intel® devcloud for oneapi. https://devcloud.intel.com/oneapi/documentation/shell-commands/.

[2] Arri A and Perry J. The f5 criterion revised[j]. *Journal of Symbolic Computation*, 46(9):1017–1029, 2001.

[3] Rodrigo Alexander Castro Campos, Feliú Davino Sagols Troncoso, and Francisco Javier Zaragoza Martínez. An efficient implementation of boolean gröbner basis computation. In *Latin American High Performance Computing Conference*, pages 116–130. Springer, 2016.

[4] Guan Yin-hua Gao Shu-hong and Volny F IV. A new incremental algorithm for computing gröbner bases[c]. *Proceedings of the 35th International Symposium on Symbolic and Algebraic ComputationNew YorkUSA*, pages 13–19, 2010.

[5] Volny F Gao Shu-hong and Wang Ming-sheng. A new algorithm for computing gröbner bases[ol].

[6] 潘森杉; 胡予濮; 王保仓. 基于标签的矩阵型 gröbner 基算法研究. **电子与信息学报**, 37(4):881–886, 2015.

[7] 狄鹏. Gröbner 基生成算法的并行 [d]. **西安电子科技大学**.