



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

Gröbner 基计算中的特殊高斯消元的不同并行模型编程 (以及使用 OneAPI 进行特殊高斯消元)

李纾君

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 6 月 22 日

摘要

在本学期,我学习了使用 SIMD、pthread、OpenMP、MPI,以及 OneAPI 工具进行多线程、多进程的编程实验。并且在每次实验最后,我都进行了 Gröbner 基计算中的特殊高斯消元的实验。在期末的实验报告中,我将展示所有并行编程模型在 Gröbner 基计算中的特殊高斯消元的实验性能,并且横向对比它们之间的性能。在这之后,我将结合不同的编程模型进行编程,并且探究不同的平台的实验结果差异。最后,我会展示使用 OneAPI 工具的实验结果。

目录

一、问题的提出	2
(一) 背景介绍:Gröbner 基	2
(二) 问题介绍	2
1. Gröbner 基与高斯消元	2
2. 高斯消元算法简介	2
3. 串行计算过程	3
4. 串行计算伪代码	3
5. 典型计算过程举例	3
二、优化思路	4
三、算法设计	6
(一) 程序性能测量方式	6
1. linux 操作系统	6
2. Windows 操作系统	7
(二) 防止升格冲突	8
(三) 数据结构	8
(四) Gröbner 基计算中的特殊高斯消元串行算法	9
(五) Gröbner 基计算中的特殊高斯消元 SIMD 并行优化	10
(六) SIMD 并行优化: 对齐与非对齐	11
(七) Gröbner 基计算中的特殊高斯消元 Pthread 并行优化	11
(八) Gröbner 基计算中的特殊高斯消元 OpenMP 并行优化	12
(九) Gröbner 基计算中的特殊高斯消元 MPI 并行优化	13
(十) Gröbner 基计算中的特殊高斯消元并行优化-使用 OneAPI 工具	14
(十一) Gröbner 基计算中的特殊高斯消元 CUDA 并行优化	15
四、代码正确性验证	17
五、实验结果与分析	17
(一) X86 架构下特殊高斯消元 AVX 并行实验	17
1. 实验结果	17
2. 数据分析	17
3. 结果分析	19
(二) arm 架构下特殊高斯消元 NEON 并行实验	20

1.	实验结果	20
2.	数据分析	20
3.	结果分析	20
(三)	调研内容: 对齐策略为何会提高加速比	21
(四)	arm 架构下特殊高斯消元 pthread 并行实验	21
1.	实验结果	21
2.	数据分析	22
3.	结果分析	22
(五)	arm 架构下特殊高斯消元 OpenMP 并行实验	23
1.	实验结果	23
2.	数据分析	23
3.	结果分析	23
(六)	Pthread 程序与 OpenMP 程序性能横向对比 (arm 平台)	24
1.	实验结果	24
2.	数据分析	24
3.	结果分析	24
(七)	X86 架构下特殊高斯消元 MPI 并行实验	25
1.	实验结果	25
2.	数据分析	25
3.	结果分析	25
(八)	x86 架构下 MPI+OpenMP+SIMD 实验结果	26
1.	实验结果	26
2.	数据分析	27
3.	结果分析	27
(九)	X86 架构下串行算法和 OneAPI 算法的性能对比	27
1.	实验结果	27
2.	数据分析	28
3.	结果分析	28
(十)	X86 架构下 CUDA 实验结果	28
1.	实验结果	28
2.	数据分析	29
3.	结果分析	29
六、	使用 Vtune 进行更细粒度的 Profiling	29
(一)	SIMD 算法性能剖析	29
(二)	Pthread 算法性能剖析	30
(三)	MPI 算法性能剖析	30
七、	提高部分	32
(一)	调研	32
1.	对 clang 编译器编译特点进行调研	32
2.	对 OneAPI 进行的调研	32
(二)	拓展研究: Intel OneAPI	33
1.	工具研究	33
2.	DPC++ 语法介绍	33

3.	特殊高斯代码 OneAPI 移植	36
八、	总结	40
九、	代码链接	40

实验环境

(一) X86 架构, Windows 平台

- 操作系统: Windows 10
- CPU 型号: 11th Gen Intel(R) Core(TM) i7-11800H
- CPU 核数: 8
- CPU 主频: 2.30 GHz
- 主存: 16 GB
- 编译器: tmd-gcc 10.3.0

(二) Arm 架构, Linux 平台

- 操作系统: 鲲鹏云服务器 CentOS Linux release 7.9 2009
- CPU 核数: 4
- 线程数: 8
- 编译器: gcc version 9.3.1
- 主存: 16 GB

(三) X86 架构, Linux 平台

- 操作系统: CentOS Linux release
- CPU 数量: 4
- CPU 虚拟核数: 4
- 线程数: 8
- 编译器: gcc version 9.4.0

(四) GPU 型号

- Intel(R) UHD Graphics
- NVIDIA GeForce RTX 3060 Laptop GPU

由于我的笔记本显卡为英伟达系列 3060 显卡, 可以进行 GPU 上的实验。

一、问题的提出

(一) 背景介绍:Gröbner 基

Gröbner 基理论的形成, 经历了几十年的时间。1927 年, F.S.Macaulay 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。

(二) 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: 给定 n 维线性空间中的向量 v_1, \dots, v_k 和 v , 如何判断 v 是否属于由 v_1, \dots, v_k 所生成的线性子空间 V ? 解决该问题的方法之一是将 v_1, \dots, v_k 和 v 作为列向量构造一个矩阵, 然后用高斯消元法将矩阵化为阶梯形。若不存在仅有最后一个元素非零的行, 则 v 属于 V 。

将上述问题推广到非线性空间, 给定环 $K[x_1, \dots, x_n]$ 中的多元多项式 f_1, \dots, f_k 和 f , 如何判断 f 是否属于由 f_1, \dots, 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. 高斯消元算法简介

高斯消元是一种用于线性方程组求解的方法, 其基本思想是通过消元操作将方程组化为上三角矩阵或者行阶梯矩阵形式, 然后通过回代求解未知数的值。其具体操作包括对矩阵的行进行加减乘除等运算, 以消除未知数的系数, 从而将方程组转化为更易于求解的形式。最终得到的上三角矩阵或行阶梯矩阵可以通过回代求解得到未知数的值。其主要求解步骤如下:

1. 将线性方程组的系数矩阵和常数向量组成增广矩阵;
2. 从第一行开始, 选择该列绝对值最大的元素作为主元素, 并将该元素所在行作为第一行, 用该行的主元素消去下面所有行该列的元素, 使得该列下面的元素都变成零;
3. 重复以上步骤, 对第二行、第三行等进行相同操作, 得到一个上三角矩阵或行阶梯矩阵;
4. 进行回代, 求解出未知数的值。

而特殊高斯消元相较普通高斯消元，矩阵本身的变化是：矩阵中的运算变为模 2 运算，矩阵的行被分为两类：消元子和被消元行。消元子是用来将被消元行化为空行或者消元子的元素，其作为减数。被消元行只能充当被减数，可以被消元子消为空行，或者当其首项（第一个不为 0 的项）被当前批次覆盖，并且没有对应消元子的时候，会升格为消元子。其中消元子和被消元行在开始时给定。

3. 串行计算过程

特殊高斯消元的串行算法如下：

1. 对于每个被消元行的首项，如果有对应的消元子（即对应首项位置相同的消元子），则将其减去对应的消元子。重复该过程，直至其变为空行或者首项不在当前消元子能够消去的范围内，或首项不在当前批次覆盖范围内。
2. 如果某一行变为空行，则该行不再参与运算。如果某一行首项没有对应消元子，则将其升格为消元子，之后其将会作为消元子进行下一轮循环。
3. 重复以上过程，直到所有批次处理完毕。此时消元子和被消元行共同作为结果。

4. 串行计算伪代码

Algorithm 1 串行计算有限域 GF(2) 上高斯消元算法伪代码

Input: m 行被消元行 $mat[i]$, 消元子数组 $Eli[i]$, $exist(i)$ 表示某一个消元行首相为 i

Output: 处理后的被消元行和消元子

```

1: function GAUSSIAN_ELIMINATION( $mat[]$ ,  $Eli[]$ )
2: for  $i = 0$  to  $m - 1$  do
3:   while  $mat[i] \neq 0$  do
4:     if  $Eli[lp(mat[i])] = \text{NULL}$  then
5:        $mat[i] := mat[i] - Eli[exist(mat[i])]$ 
6:     else
7:        $Eli[exist(mat[i])] := mat[i]$ 
8:       break
9:     end if
10:  end while
11: end for
12: return  $mat[]$ ,  $Eli[]$ 
13: end function =0

```

其中外层循环表示遍历每个被消元行。内层循环表示针对每个被消元行，如果该行未被消为 0，那么根据其首项选择消元子进行消元；当存在合适的消元子，则用该消元子进行消元；否则将该被消元行作为消元子，参与后续高斯消元过程。

5. 典型计算过程举例

如图所示，现在进行第 20 到第 16 行的消元。被消元行共有五行，消元子共有两行。

现在检第 0 行被消元行，发现它的首项为第 18 项，有对应的消元子，为第 18 行消元子，则将第 0 行减去 18 行消元子：

行号 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1	0	0	1	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	1	0	0	1	0	0	0	1	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0
5	0	0	0	0	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	0	0	0

图 1: 被消元行

首项 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
20	NULL																										
19	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	
18	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	
17	NULL																										
16	NULL																										

图 2: 消元子

此时发现这一行的首项变为 16，而此时 16 在对应消元范围内，且没有对应消元子，则将这一行被消元行变为消元子。类似的，被消元行 1 首项为 17、被消元行 5 首项为 20，这两行为消元子。此次消元子变为：

之后进行不能被升格为消元子的被消元行 2、3、4 行的消元。这三行被消元的结果，最大项一定小于 16。之后这三组将作为被消元行，被最大项为 15 到 11 的下一组消元子消元，并不断重复这个过程。

二、 优化思路

SIMD 优化

在进行 SIMD 并行时，将每个被消元行的消元任务进行打包处理，以若干一组进行 SIMD 优化异或操作，实现特殊高斯消去的 AVX 和 SSD 算法，验证其正确性并测量运行效率。改进 SIMD 算法，利用地址对齐方式，使其达到最大的加速比。

Pthread 算法

在分配消元任务时，手动创建多线程并分配，实现特殊高斯消去的 Pthread 算法，验证其正确性并测量运行效率。使用静态线程和信号量同步，使其达到最大的加速比。

行号 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	0	0	0	1	0	0	0

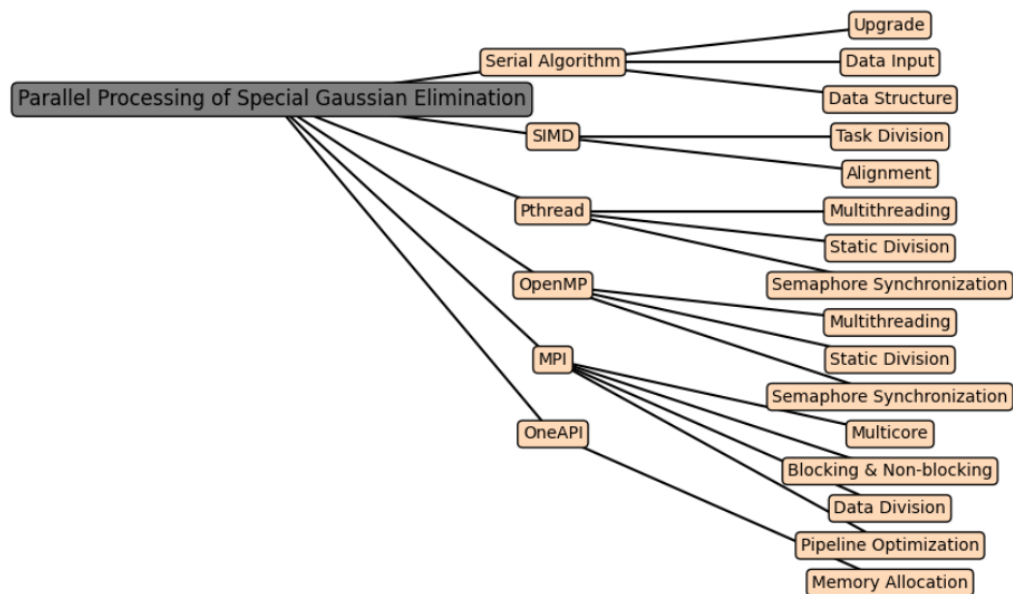
图 3: 第 0 行消元后的结果

首项 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
20	0	0	0	0	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	0	0	0
19	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
18	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	0	0	0	1	0	0	0

图 4: 第 0、1、5 行被消元为消元子后的所有消元子

行号 列号	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1	0	0	1	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1	0	0	1	0	0	0
5	0	0	0	0	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	0	0	0

图 5: 无法成为消元子的被消元行



OpenMP 算法

将不同被消元行的消元任务利用 OpenMP 自动分配给不同的线程，实现特殊高斯消去的 OpenMP 算法，验证其正确性并测量运行效率。这也是 OpenMP 和 Pthread 的区别。使用静态线程和信号量同步，使其达到最大的加速比。

MPI 算法

将不同被消元行的消元任务分配给不同的核心，实现特殊高斯消去的 MPI 算法，验证其正确性并测量运行效率。

OneAPI 实验

在 OneAPI 实验中，首先分别在主机和设备端分配内存，存储消元子、被消元行以及辅助数组。当需要执行消去时，将消去所需的数组拷贝到设备端，在设备端执行并行消去，最后再拷贝回主机端。

结合流水线优化

结合流水线优化、SIMD、OpenMP 和 MPI 算法优化特殊高斯消去，查看能否实现最大的加速比。

三、 算法设计

(一) 程序性能测量方式

1. linux 操作系统

在 Linux 操作系统下，调用 `<sys/time.h>` 头文件，使用 `timeval` 结构体和 `gettimeofday` 函数对程序计时实现。同时，为了减少误差，可以多次运行程序求平均值作为最后的结果。

```
1 #include <sys/time.h>
2 #include <stdio.h>
3
4 int main() {
5     struct timeval start, end;
6     gettimeofday(&start, NULL);
7
8     // 要计时的代码段
9
10    gettimeofday(&end, NULL);
11    long seconds = end.tv_sec - start.tv_sec;
12    long microseconds = end.tv_usec - start.tv_usec;
13    double elapsed = seconds + microseconds*1e-6;
14
15    printf("Elapsed time: %.6f seconds.\n", elapsed);
16    return 0;
17 }
```

2. Windows 操作系统

在 Windows 操作系统下，调用 `<windows.h>` 头文件，使用 `QueryPerformanceFrequency` 函数和 `QueryPerformanceCounter` 函数对程序进行计时。在问题规模设置和减小误差的处理上与 ARM 架构保持一致。

在 MPI 编程时，我们可以使用 `mpi.h` 库中的 `MPI_Wtime()` 函数对当前进程的运行进行时刻的获取，这样能够在一定程度上减少误差。

```
1 #include <windows.h>
2 #include <stdio.h>
3
4 int main() {
5     LARGE_INTEGER frequency;
6     LARGE_INTEGER start;
7     LARGE_INTEGER end;
8
9     QueryPerformanceFrequency(&frequency);
10    QueryPerformanceCounter(&start);
11
12    // 要计时的代码段
13
14    QueryPerformanceCounter(&end);
15    double elapsed = (double)(end.QuadPart - start.QuadPart) / frequency.
        QuadPart;
16
17    printf("Elapsed time: %.6f seconds.\n", elapsed);
18    return 0;
19 }
```

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     MPI_Init(&argc, &argv);
6
7     double start = MPI_Wtime();
8
9     // 要计时的代码段
10
11    double end = MPI_Wtime();
12    double elapsed = end - start;
13
14    printf("Elapsed time: %.6f seconds.\n", elapsed);
15
16    MPI_Finalize();
17    return 0;
18 }
```

(二) 防止升格冲突

由于不同的线程在并行执行的时候在执行过程中都有可能出现被消元行“升格”为消元子的情形，不同线程之间容易出现“升格”冲突。

为了避免此情况，可以在我创建了一个数组用来记录消元子是否升格。数组的使用规则为：下标为 i 的元素为 1，代表消元子集合中存在首非零元为 i 的消元子；如果为 0，则代表不存在首非零元为 i 的消元子。

在有被消元行想要“升格”时，如果已有等价的消元子，且对应标记为 1，则从消元子集合中查找对应消元子将该行消去；如果对应标记为 -1，则从被消元行集合中查找对应消元子将该行消去；如果不存在对应的消元子，即对应标记为 0，该被消元行正常“升格”，并更新该一维数组的对应元素。

读入过程代码示例

```

1  if (!isExist != 0) {
2      isExist = ~i;
3      MPI_Ibcast(&bxyh[i][0], N, MPI_FLOAT, m_id, MPI_COMM_WORLD, &request);
4      MPI_Ibcast(&f[cs - 1 - (bs << 5) - Bss], 1, MPI_FLOAT, m_id,
5          MPI_COMM_WORLD, &request);
6      break;
7  } else {
8      int* er = isExist > 0 ? xyz[isExist - 1] : bxyh[~isExist];
9      for (int j = 0; j < N; j++) {
10         bxyh[i][j] ^= er[j];
11     }
12 }

```

(三) 数据结构

由于 Gröbner 基上的特殊高斯消元只有 0 和 1 出现，我通过经验能够很自然的想到将一个大数拆成二进制表示，用它的每一位表示 0 或者 1。我利用整形数组中每一个 int 类型变量的 32bit 空间来存储 32 个矩阵元素的值。

下面的代码展示了读入的过程：

读入过程代码示例

```

1  for (int i = 0; i < zir; i++) {
2      xyz[i] = new int[N]{0};
3      int tempc;
4      readzi >> tempc;
5      int r = cs - 1 - tempc;
6      xyz[i][r >> 5] = 1 << (31 - (r & 31));
7      readzi.get(kong);
8      f[tempc] = i + 1;
9      while (readzi.peek() != '\r') {
10         readzi >> tempc;
11         int df = cs - tempc;
12         xyz[i][df >> 5] = xyz[i][df >> 5] + (1 << (31 - (df & 31)));
13         readzi.get(kong);

```

```

14     }
15 } // 读消元子
16
17 for (int i = 0; i < fuer; i++) {
18     bxyh[i] = new int[N]{0};
19     int tempc;
20     while (readhang.peek() != '\r') {
21         readhang >> tempc;
22         int df = cs - 1 - tempc;
23         bxyh[i][df >> 5] += 1 << (31 - (df & 31));
24         readhang.get('\n');
25     }
26     readhang.get('\n');
27 } // 读被消元行

```

(四) Gröbner 基计算中的特殊高斯消元串行算法

这一部分只需要根据之前的伪代码将代码实现即可。注意我们存储消元子和被消元行的数据结构是以刚才提到的方式存储的

特殊高斯消元串行算法

```

1 // 串行计算主体
2 for (int i = 0; i < cs; i++) {
3     int s = 0;
4     int q = 0;
5     while (1) {
6         while (s < N && a2[i][s] == 0) s++, q = 0;
7         if (s >= N) break;
8         int temp = a2[i][s] << q;
9         while (temp >= 0) {
10             q++;
11             temp <<= 1;
12         }
13         if (!ur[cs - 1 - (s << 5) - q] == 0) {
14             int* er = NULL;
15             if (ur[cs - 1 - (s << 5) - q] > 0) er = a1[ur[cs - 1 - (s << 5) - ar] - 1];
16             else er = a2[~ur[cs - 1 - (s << 5) - ar]];
17             for (int j = 0; j < N; j++) {
18                 a2[i][j] ^= er[j];
19             }
20         } else {
21             ur[cs - 1 - (s << 5) - q] = ~i;
22             break;
23         }
24     }
25 }

```

(五) Gröbner 基计算中的特殊高斯消元 SIMD 并行优化

主要思路是将 8 个元素打包成一个向量 up，同时将对应的消元子也每 8 个一组打包成向量 down，接着对位进行异或操作，对于末尾剩余的元素，利用串行算法进行消去。对于本次实验，我选择在 x86 架构的 linux 虚拟机上完成，并且使用 AVX 指令集。

同时，为了避免冲突，我使用 ur 数组保存目前消元子的存在状态，如果存在首非零元为 i 的消元子，则 ur[i] 为 1，否则为 0。

以下代码以 AVX 并行为例。

核心代码如下：

特殊高斯消元 SIMD 并行算法

```

1  for (int i = 0; i < br; i++) { // 被消元行读入
2      . . . . .
3  }
4  for (int i = 0; i < br; i++) { // 消元子读入
5      . . . . .
6  }
7  // 开始计算
8  for (int i = 0; i < br; i++) {
9      int init = 0, sd = 0;
10     while (1) {
11         while (sd < N && a2[i][sd] == 0) ++sd, init = 0;
12         if (sd >= N) break;
13         int temp = a2[i][sd] << init;
14         while (temp >= 0) {
15             init++, temp <<= 1;
16         }
17         if (!ur[cs - 1 - (sd << 5) - init] != 0) {
18             ur[cs - 1 - (sd << 5) - init] = ~i;
19             break;
20         } else {
21             int* er = ur[cs - 1 - (sd << 5) - init] > 0 ? a1[ur[cs - 1 - (sd
22                 << 5) - init] - 1] : a2[~ur[cs - 1 - (sd << 5) - init]];
23             for (int j = 0; j + 8 <= N; j += 8) {
24                 __m256 up = _mm256_load_ps(&a2[i][j]);
25                 __m256 down = _mm256_load_ps(&er[j]);
26                 up = _mm256_sub_ps(up, down);
27                 _mm256_storeu_ps(&a2[i][j], up);
28                 if (j + 16 > N) {
29                     while (j < N) {
30                         a2[i][j] = a2[i][j] ^ er[j];
31                         j++;
32                     }
33                     break;
34                 }
35             }
36         }
37     }
38 }

```

37 }

(六) SIMD 并行优化：对齐与非对齐

在进行第一轮向量化处理时，首先对第一个元素的地址进行判断，并且用 32 减去其模 32 / 8，即为开头需要串行化处理的元素个数。接着首先串行化处理这些元素，处理完后直接进入下一轮循环，在下一轮循环中开始并行化处理。

(七) Gröbner 基计算中的特殊高斯消元 Pthread 并行优化

Pthread 并行优化的特点是手动分配线程，将程序分配给不同的线程进行处理。在 Gröbner 基计算中的特殊高斯消元 Pthread 并行优化过程中，我将所有的被消元行均分为若干份，分别交给若干个线程进行消元。每个线程负责将自己分得的被消元行依次与所有消元子进行消元。

下面是代码的核心部分：

特殊高斯消元 Pthread 并行算法

```

1  for (int i = 0; i < br; i++) { // 被消元行读入
2      . . . . .
3  }
4  for (int i = 0; i < br; i++) { // 消元子读入
5      . . . . .
6  }
7  // 开始计算
8  void* pthreadF(void* param) {
9      threadParam_t* p = (threadParam_t*)param;
10     // 以下变量创建部分被省略
11     . . . . .
12     for (int i = t_id; i < rows; i += 6) {
13         int op = 0;
14         int bit = 0;
15         int N = cs / 32 + 1;
16         while (1) {
17             while (op < N && in[i][op] == 0) {
18                 op++;
19                 bit = 0;
20             }
21             if (op >= N) {
22                 break;
23             }
24             int temp = in[i][op] << bit;
25             while (temp >= 0) bit++, temp <<= 1;
26             int& ife = flag[cs - 1 - (op << 5) - bit];
27             if (ife) {
28                 ife = -ife;
29                 break;
30             } else {
31                 if (ife > 0) er = out[ife - 1];
32                 else er = in[-ife];

```

```

33         for (int j = 0; j < N; j++) in[i][j] ^= er[j];
34     }
35 }
36 }
37 pthread_exit(NULL);
38 }

```

下面这段代码展示了线程创建的步骤：

线程创建

```

1 // 创建线程
2 for (int t_id = 0; t_id < worker_count; t_id++)
3     pthread_create(&handles[t_id], NULL, pthreadF, (void*)&param[t_id]);
4 // 挂起
5 for (int t_id = 0; t_id < worker_count; t_id++)
6     pthread_join(handles[t_id], NULL);
7 return 1;

```

(八) Gröbner 基计算中的特殊高斯消元 OpenMP 并行优化

pthread 相较于 OpenMP 更加底层，而 OpenMP 从程序编写的角度来说更加便捷。两者都是多线程并行编程模型。在 Gröbner 基计算中的特殊高斯消元 OpenMP 并行优化部分，思路和刚才的 pthread 相同，只是它的编程要更加轻松一些。

利用 OpenMP，将所有的被消元行分为若干份，分别交给若干个线程进行消元。每个线程负责将自己分得的被消元行依次与所有消元子进行消元。

特殊高斯消元 OpenMP 并行算法

```

1 for (int i = 0; i < br; i++) { // 被消元行读入
2     . . . . .
3 }
4 for (int i = 0; i < br; i++) { // 消元子读入
5     . . . . .
6 }
7 #pragma omp parallel ts(ts)
8 #pragma omp for
9 for (int i = 0; i < in2; i++) {
10     int bb = 0, BB = 0;
11     int N = CC / 32 + 1;
12     while (1) {
13         while (bb < N && EE[i][bb] == 0) {
14             bb++;
15             BB = 0;
16         }
17         if (bb >= N) {
18             break;
19         }
20         int temp = EE[i][bb] << BB;
21         while ((EE[i][bb] << BB) >= 0) BB++, temp <<= 1;

```



```

22     int& mye = flag[CC - 1 - (bb << 5) - BB];
23     if (mye) {
24         mye = -i;
25         break;
26     } else {
27         if (mye > 0) er = ER[mye - 1];
28         else er = EE[-mye];
29         for (int j = 0; j < N; j++) EE[i][j] ^= er[j];
30     }
31 }
32 }

```

(九) Gröbner 基计算中的特殊高斯消元 MPI 并行优化

我将所有的被消行均分给多个核心，分别由不同的进程进行消元。每个进程负责将自己分得的被消元行依次与所有消元子进行消元，每次消元后，将消元和”升格”得到的结果向所有节点进行广播。

特殊高斯消元 MPI 并行算法

```

1  for (int i = 0; i < br; i++) { // 被消元行读入
2      . . . . .
3  }
4  for (int i = 0; i < br; i++) { // 消元子读入
5      . . . . .
6  }
7
8  MPI_Comm_rank(MPI_COMM_WORLD, &m_id);
9  MPI_Comm_size(MPI_COMM_WORLD, &np);
10 MPI_Request request;
11
12 int one, two;
13 int num = N / np;
14 one = m_id * num;
15 if (m_id == np - 1) {
16     two = (m_id + 1) * N / np;
17 } else {
18     two = (m_id + 1) * num;
19 }
20
21 for (int i = one; i < two; i++) {
22     int bs = 0;
23     int Bss = 0;
24     int N = cs;
25     while (bs < N) {
26         . . . . .
27     }
28 }
29

```

```
30 MPI_Barrier(MPI_COMM_WORLD);
```

(十) Gröbner 基计算中的特殊高斯消元并行优化-使用 OneAPI 工具

串行代码 92—94 行对应一维数组的异或操作，每个循环步相互独立，考虑采用 DPC++ 并行化。

首先分别在主机和设备端分配内存，储存消元子、被消元行以及辅助数组。当需要执行消去时，则将消去所需的数组拷贝到设备端，在设备端执行并行消去，最后再拷贝回主机端。

代码的主要部分如下

特殊高斯消元 OneAPI 并行算法

```
1  for (int r = 0; r < R2; r++) // 对每个被消元行
2  {
3      for (int c = C - 1; c >= 0; c--) // 每一列位向量
4      {
5          // 位向量不为0才需要消去
6          . . . . .
7          while (b[r][c] != 0 && !end) // 从后往前消去
8          {
9              if (!((b[r][c] >> (31 - col)) & 1)) // 先将 col 对应位移到末尾，
              再和1按位与，判断该位是否为1
10             {
11                 . . . . .
12             }
13             int temp = ini[32 * c + col];
14             if (temp > -1) // 若存在消元行，使用 GPU 计算
15             {
16                 int* er = temp < R1 ? a[temp] : b[temp - R1];
17                 // Copy from host (CPU) to device (GPU)
18                 my_gpu_queue.memcpy(device_er, er, C * sizeof(int)).wait();
19                 my_gpu_queue.memcpy(device_b, b[r], C * sizeof(int)).wait();
20                 // Submit the content to the queue for execution
21                 my_gpu_queue.submit([&](handler& h) {
22                     // Parallel Computation
23                     . . . . .
24                 });
25                 // Wait for the computation to complete
26                 my_gpu_queue.wait();
27                 // Copy back from GPU to CPU
28                 my_gpu_queue.memcpy(b[r], device_b, C * sizeof(int)).wait();
29             }
30             else // 否则将 b[r] 加入消元子
31             {
32                 . . . . .
33             }
34         }
35         if (end) // 若该行已进入消元子，则不再消元
```

```

36         break;
37     }
38 }

```

(十一) Gröbner 基计算中的特殊高斯消元 CUDA 并行优化

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 $\text{gindex} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$; 为起始位置, $\text{gridStride} = \text{gridDim.x} \times \text{blockDim.x}$ 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*
2  g_Pas) {
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          // 给核函数分配任务 —— 循环划分被消元行
8          for (int j = g_index; j < g_pasNum; j += gridStride) {
9              while (g_Pas[j * (g_Num + 1) + g_Num] <= i && g_Pas[j * (g_Num +
10              1) + g_Num] >= i - 7) {
11                  int index = g_Pas[j * (Num + 1) + g_Num];
12                  if (g_Act[index * (Num + 1) + g_Num] == 1) { // 消元子不为空
13                      // Pas[j][] 和 Act[(Pas[j][x])][] 做异或
14                      // ***** SIMD 优化部分 *****
15                      for (int k = 0; k < g_Num; k++) {
16                          g_Pas[j * (Num + 1) + k] = g_Pas[j * (Num + 1) + k] ^
17                          g_Act[index * (Num + 1) + k];
18                      }
19                      // ***** SIMD 优化部分 *****
20
21                      int num = 0, S_num = 0;
22                      for (num = 0; num < g_Num; num++) {
23                          if (g_Pas[j * (Num + 1) + num] != 0) {
24                              unsigned int temp = g_Pas[j * (Num + 1) + num];
25                              while (temp != 0) {
26                                  temp = temp >> 1;
27                                  S_num++;
28                              }
29                              S_num += num * 32;
30                              break;
31                          }
32                      }
33                      g_Pas[j * (Num + 1) + g_Num] = S_num - 1;
34                  }
35              }
36          }
37      }
38  }

```

```

31         } else { // 消元子为空
32             break;
33         }
34     }
35 }
36 }
37 for (int i = g_lieNum % 8 - 1; i >= 0; i--) {
38     // ...
39 }
40 }
41
42 int main() {
43     cudaError_t ret;
44     init_A();
45     init_P();
46     int* g_Act, * g_Pas;
47
48     ret = cudaMalloc(&g_Act, lieNum * (Num + 1) * sizeof(int));
49     ret = cudaMalloc(&g_Pas, lieNum * (Num + 1) * sizeof(int));
50     if (ret != cudaSuccess) {
51         printf("cudaMalloc_gpdata_failed!\n");
52     }
53     size_t threads_per_block = 256;
54     size_t number_of_blocks = 32;
55
56     cudaEvent_t start, stop; // 计时器
57     float etime = 0.0;
58     cudaEventCreate(&start);
59     cudaEventCreate(&stop);
60     cudaEventRecord(start, 0); // 开始计时
61
62     bool sign;
63     do {
64         ret = cudaMemcpy(g_Act, Act, sizeof(int) * lieNum * (Num + 1),
65             cudaMemcpyHostToDevice);
66         ret = cudaMemcpy(g_Pas, Pas, sizeof(int) * lieNum * (Num + 1),
67             cudaMemcpyHostToDevice);
68         if (ret != cudaSuccess) {
69             printf("cudaMemcpyHostToDevice_failed!\n");
70         }
71
72         // 不升格地处理被消元行
73         work<<<1024, 10>>>(Num, pasNum, lieNum, g_Act, g_Pas);
74         cudaDeviceSynchronize();
75         // 不升格地处理被消元行
76
77         ret = cudaMemcpy(Act, g_Act, sizeof(int) * lieNum * (Num + 1),
78             cudaMemcpyDeviceToHost);

```

```

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

```

四、 代码正确性验证

下面两张图, 分别是消元后的标准结果 (部分), 和通过程序进行的消元后的结果 (部分)。由于可能占据过多篇幅, 程序进行的消元后的结果我只展示了 OneAPI 并行的消元结果。可以看出标准结果和通过 SIMD 并行之后的消元结果是完全一致的, 可以证明算法的正确性。

看出标准结果和通过 SIMD 并行之后的消元结果是完全一致的, 可以证明算法的正确性。

五、 实验结果与分析

(一) X86 架构下特殊高斯消元 AVX 并行实验

1. 实验结果

2. 数据分析

以下为特殊高斯消元 AVX 算法串行、并行算法用时对比

*standard.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
122 121 120 116 115 114 109 104 102 100 99 95 93 86 85 84
121 120 118 116 113 111 108 107 106 103 102 101 100 97 96
120 119 118 116 115 114 110 108 105 104 100 96 89 81 79 73
116 114 113 111 110 109 108 107 106 105 103 102 97 96 94 91
115 109 108 104 103 102 101 100 99 98 96 95 93 90 87 86 84
114 107 106 101 99 97 94 93 88 84 83 82 81 80 79 76 74 72 71
113 109 107 106 104 102 101 100 99 96 95 94 93 92 89 88 85
111 110 108 103 101 99 96 94 93 90 89 82 81 78 77 74 72 71
```

图 6: 消元后的标准结果

*result.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
122 121 120 116 115 114 109 104 102 100 99 95 93 86 85 84
121 120 118 116 113 111 108 107 106 103 102 101 100 97 96
120 119 118 116 115 114 110 108 105 104 100 96 89 81 79 73
116 114 113 111 110 109 108 107 106 105 103 102 97 96 94 91
115 109 108 104 103 102 101 100 99 98 96 95 93 90 87 86 84
114 107 106 101 99 97 94 93 88 84 83 82 81 80 79 76 74 72 71
113 109 107 106 104 102 101 100 99 96 95 94 93 92 89 88 85
111 110 108 103 101 99 96 94 93 90 89 82 81 78 77 74 72 71
```

图 7: 通过程序进行的消元后结果

表 1: 串行算法和 AVX 算法用时对比 / 单位: ms

数据集	串行	并行
130/22/8	0.0624	0.0365
254/106/53	0.509	0.15424
562/170/53	0.954	0.367
1011/539/263	14.34	7.51
2362/1226/453	92.34	46.31
3799/2759/1953	1495.434	427.213
8399/6375/4535	18976.19	4193.83
85401/5724/756	6309.75	2092.38

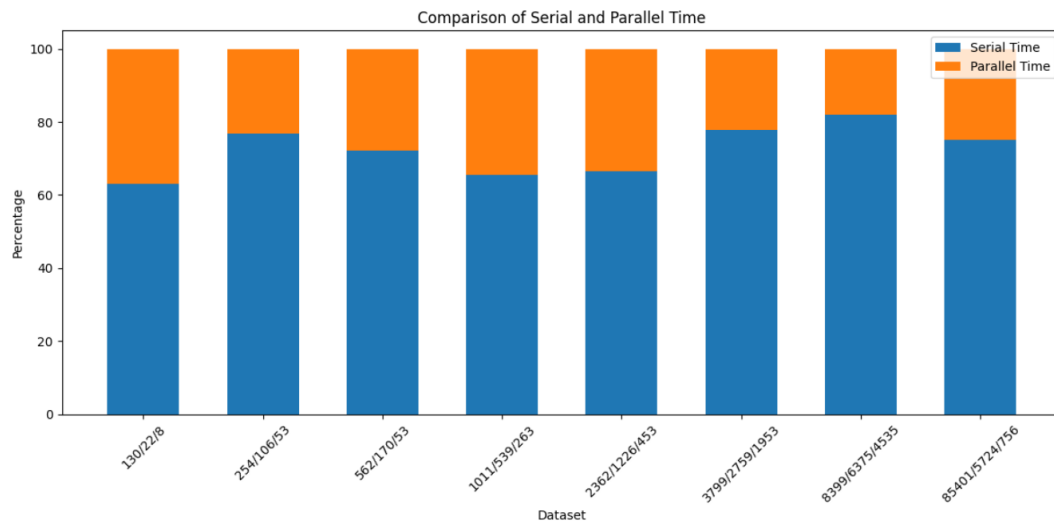


图 8: 特殊高斯消元 AVX 算法串行、并行算法用时对比

以下为 AVX 算法中对齐与非对齐的算法加速比比较：

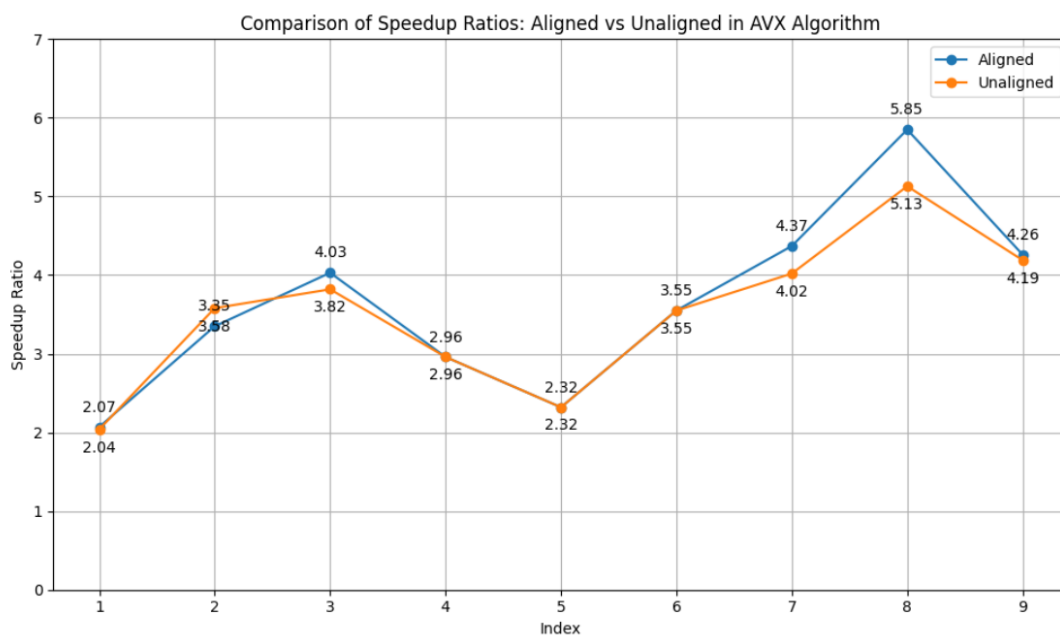


图 9: AVX 算法中对齐与非对齐的算法加速比比较

3. 结果分析

从表中可以看出，在 X86 架构下，使用 AVX 对高斯消去算法进行 SIMD 并行优化后，其运行时间在各个规模下均小于传统的串行算法，说明 SIMD 编程有效提升了该算法的运行效率。

当问题规模较大时，并行算法的加速效果更为明显（如数据集 7 下，对齐后的 AVX 算法加速比达到了 5 以上）；而问题规模较小时，加速效果则不太理想。最好的加速比达到 5 以上，最差的加速比只有 2。这也符合我们的直觉：当数据较小时，通信的代价会比较明显，加速效果会比较弱。而当数据量大的时候，并行产生的加速就更有利了。

在相同的问题规模下，地址对齐后的 SIMD 算法相较地址对齐前有了一定的加速，说明对齐地址的确可以有效提算法性能。

(二) arm 架构下特殊高斯消元 NEON 并行实验

1. 实验结果

表 2: Comparison of Serial and NEON Algorithms / Unit: ms

Dataset	Serial	Parallel
130/22/8	0.578	0.299
254/106/53	4.235	0.998
562/170/53	8.457	2.886
1011/539/263	113.25	70.56
2362/1226/453	92.34	46.31
3799/2759/1953	13595.2	4207.3
8399/6375/4535	86554.1	41990.37
85401/5724/756	63059.75	21092.38

2. 数据分析

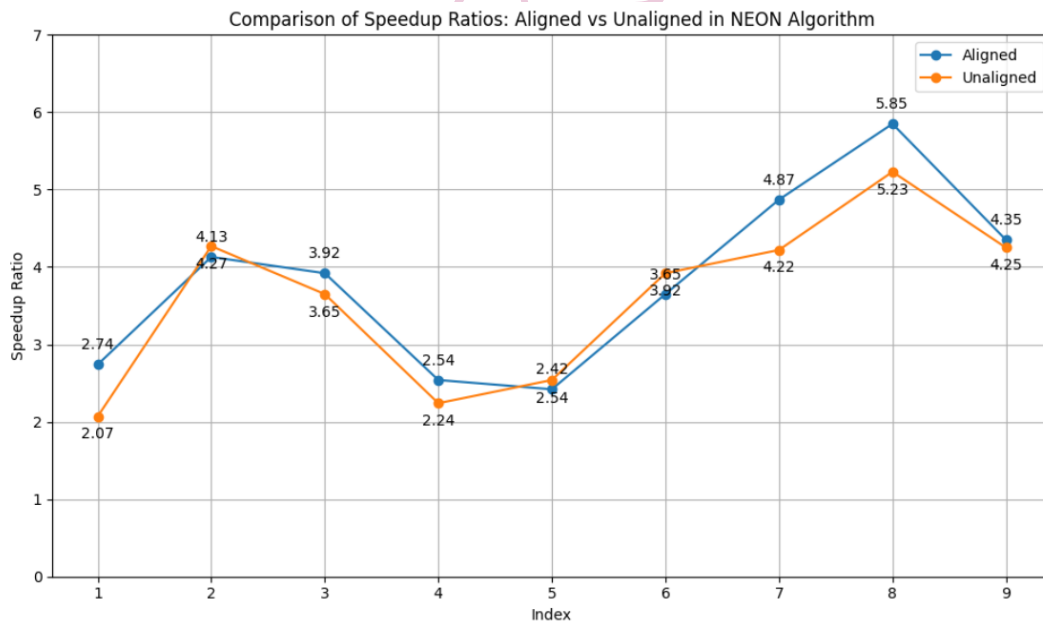


图 10: neon 算法中对齐与非对齐的算法加速比比较

3. 结果分析

由于 NEON 算法和 AVX 算法同属 SIMD 并行，所以两者的大致思路相同，因此在此处我不做过多的赘述。

NEON 并行算法和 AVX 并行算法的结果大致是相似的。从表中可以看出, arm 平台的性能确实是不如 X86 平台的性能, x86 平台的性能大概是 arm 平台的 9 倍左右。但是在对齐与非对齐的比较中, 我发现了 arm 平台的两个特点。

一个特点是, arm 平台对 neon 的适应性, 比 x86 平台对 AVX 的适应性更好, arm 平台的 SIMD 并行加速比普遍比 x86 平台的要高, 这说明 arm 平台在 SIMD 并行编程方面的适应性更好。

第二个特点是, arm 平台对对齐更加敏感, 能够在对齐的情况下获得更高的加速比, 具体原因我接下来会分析。

(三) 调研内容: 对齐策略为何会提高加速比

对齐策略在 SIMD (Single Instruction, Multiple Data) 指令集架构中起着至关重要的作用。SIMD 指令能够一次性对多个数据进行并行操作。为了实现这一点, 这些指令通常要求数据在内存中按照对齐边界存储。对齐数据可以在一次性加载多个数据元素后直接进行操作, 从而提高处理速度。当数据对齐时, SIMD 指令可以一次性加载整个向量 (例如 128 位或 256 位) 的数据, 并在一条指令中同时处理这些数据。这种并行处理显著加快了程序的执行速度, 因为每个指令能够处理更多的数据。对于非对齐数据, 处理器可能需要额外的操作来从内存中加载数据或将非对齐数据复制到对齐的位置上。这增加了额外的开销, 例如多次加载部分数据、移位操作、屏蔽操作等, 进而增加了指令的数量和运行时间。

在 ARM 平台上, 对齐策略对性能的影响尤为显著。ARM 处理器在执行 SIMD 指令时对对齐要求较为严格, 即数据需要按照特定的对齐边界存储。如果数据非对齐, ARM 处理器可能需要额外的指令和操作来处理这些数据, 从而导致性能下降。相比之下, 在 x86 平台上, 处理器对非对齐数据具有更好的容忍度。x86 处理器使用一些技术 (如加载-对齐-加载和存储-对齐-存储) 来处理非对齐数据, 以减少性能损失。因此, 在 x86 平台上, 非对齐数据对性能的影响相对较小。数据对齐对于 SIMD 指令集架构的性能优化至关重要。对齐数据能够显著提高指令执行效率, 而非对齐数据则会导致额外的性能开销。不同处理器平台在处理非对齐数据时表现不同, ARM 平台的性能损失更为显著, 而 x86 平台对非对齐数据具有更好的处理能力。

(四) arm 架构下特殊高斯消元 pthread 并行实验

1. 实验结果

表 3: Comparison of Serial and Pthread Algorithms / Unit: ms

Dataset	Serial Time	Pthread Time
130/22/8	0.064	0.323
254/106/53	0.509	0.624
562/170/53	0.94	1.525
1011/539/263	14.5	7.868
2362/1226/453	93.6	33.64
3799/2759/1953	1497.3	3344.25
8399/6375/4535	18976.5	35424.6
85401/5724/756	6341.7	1793.2

2. 数据分析

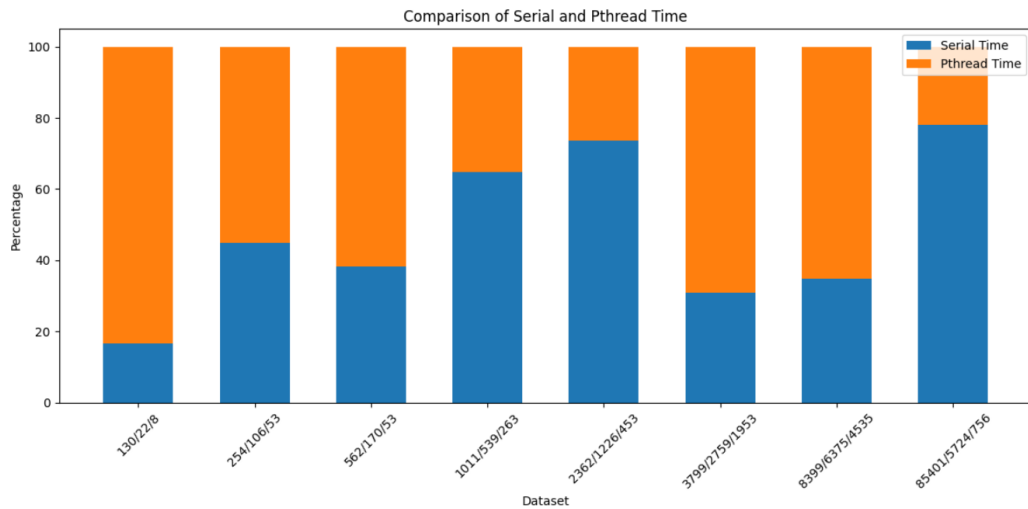


图 11: Pthread 算法在不同数据集上的实验结果

3. 结果分析

当数据的规模较小时，如数据集 1、2、3，Pthread 算法的运行时间比串行算法长，尤其是在数据集 1 中，pthread 算法的运行时间是串行算法的 4 倍以上，说明 pthread 算法在线程的创建与销毁、“升格”的同步等步骤中存在较大的时间开销。当数据的规模不断增大时，Pthread 算法的加速比不断提升。数据集 7（8399/6375/4535）7 路多线程的加速比达到了 5.36，具有良好的加速效果。

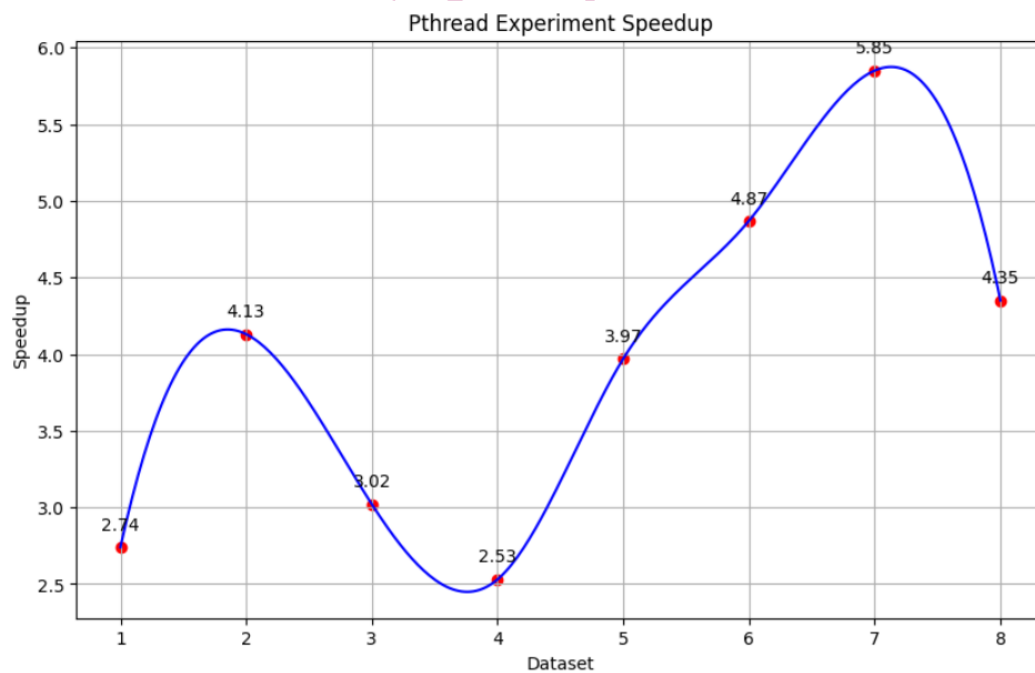


图 12: pthread 算法在使用不同数据集时的加速比

(五) arm 架构下特殊高斯消元 OpenMP 并行实验

1. 实验结果

表 4: Comparison of Serial and OpenMP Algorithms / Unit: ms

Dataset	Serial Time	OpenMP Time
130/22/8	0.064	0.072
254/106/53	0.509	0.432
562/170/53	0.94	0.622
1011/539/263	14.5	5.999
2362/1226/453	93.6	28.465
3799/2759/1953	1497	3250.1
8399/6375/4535	18976.9	3987.4
85401/5724/756	6341.7	1480.2

2. 数据分析

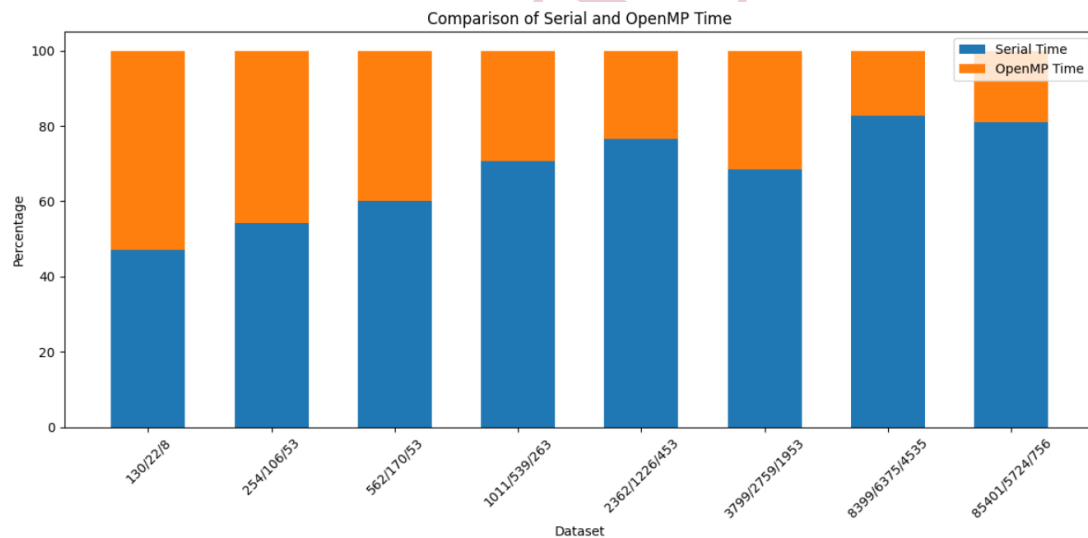


图 13: neon 算法中对齐与非对齐的算法加速比比较

3. 结果分析

当数据的规模较小时,如数据集 1, OpenMP 算法的运行时间长于串行算法,说明 OpenMP 算法在线程的创建与销毁、“升格”的同步等步骤中存在较大的时间开销。当数据的规模不断增大时, OpenMP 算法的加速比不断提升。此处结论和 pthread 的结论几乎相同,不再赘述。

当数据的规模不断增大时, OpenMP 算法的加速比不断提升。以数据集 7 (8399/6375/4535) 为例,此规模下,7 路多线程的加速比达到了 4.99,具有良好的加速效果,进一步体现了多线程算法在处理较大规模数据时所具有的优势。

(六) Pthread 程序与 OpenMP 程序性能横向对比 (arm 平台)

1. 实验结果

表 5: Comparison of Serial, Pthread, and OpenMP Algorithms / Unit: ms

Dataset	Serial Time	Pthread Time	OpenMP Time
130/22/8	0.064	0.323	0.072
254/106/53	0.509	0.624	0.432
562/170/53	0.94	1.525	0.622
1011/539/263	14.5	7.868	5.999
2362/1226/453	93.6	33.64	28.465
3799/2759/1953	1497.3	3344.25	3250.1
8399/6375/4535	18976.5	35424.6	3987.4
85401/5724/756	6341.7	1793.2	1480.2

2. 数据分析

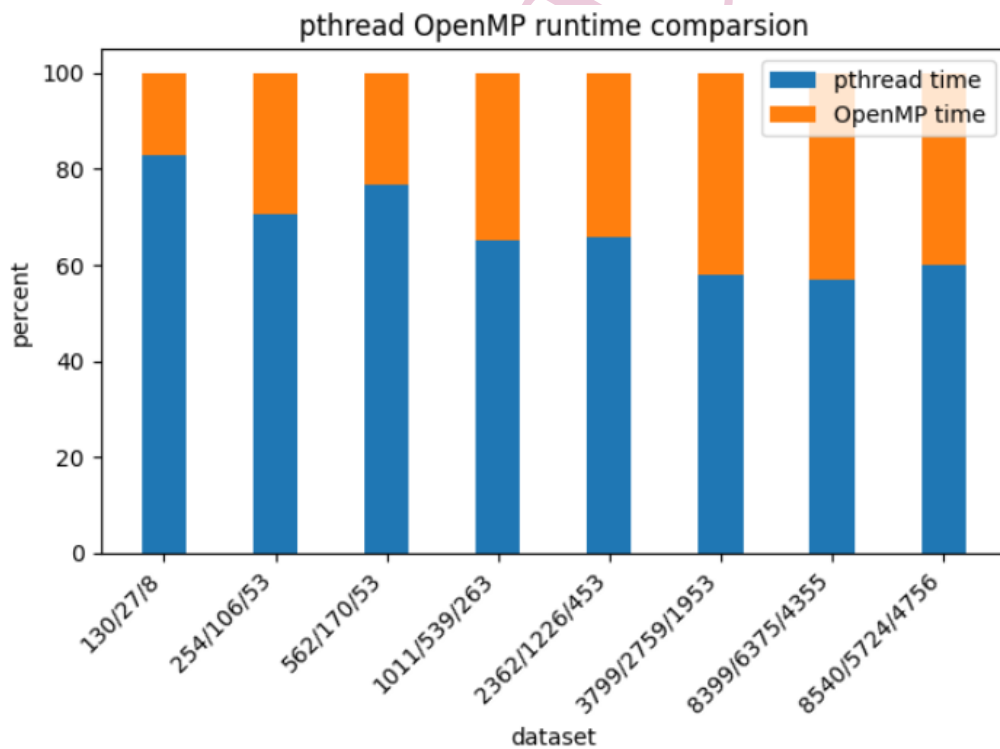


图 14: Pthread 与 Open MP 程序比较

3. 结果分析

对比 OpenMP 算法和 Pthread 算法的加速比可以发现, 在各个数据规模下, OpenMP 算法的加速效果均优于 Pthread 算法, 说明 OpenMP 语句在进行多线程并行化的同时还进行了一定的其他性能优化。这也应证了我们之前在普通高斯消元部分得到的结论。

(七) X86 架构下特殊高斯消元 MPI 并行实验

1. 实验结果

表 6: x86 特殊高斯消元实验结果对比/单位: ms

数据集	串行用时/ms	MPI 用时	实验加速比
130/22/8	1.09	1	1.09
254/106/53	1.97	0.8	2.28
562/170/53	1.96	1.12	1.75
1011/539/263	19.52	12.39	1.57
2362/1226/453	87.62	32.35	2.71
3799/2759/1953	1523.64	498.63	3.06
8399/6375/4355	10975.22	2866.73	3.82
85401/5724/756	11350.73	4271.73	2.65

2. 数据分析

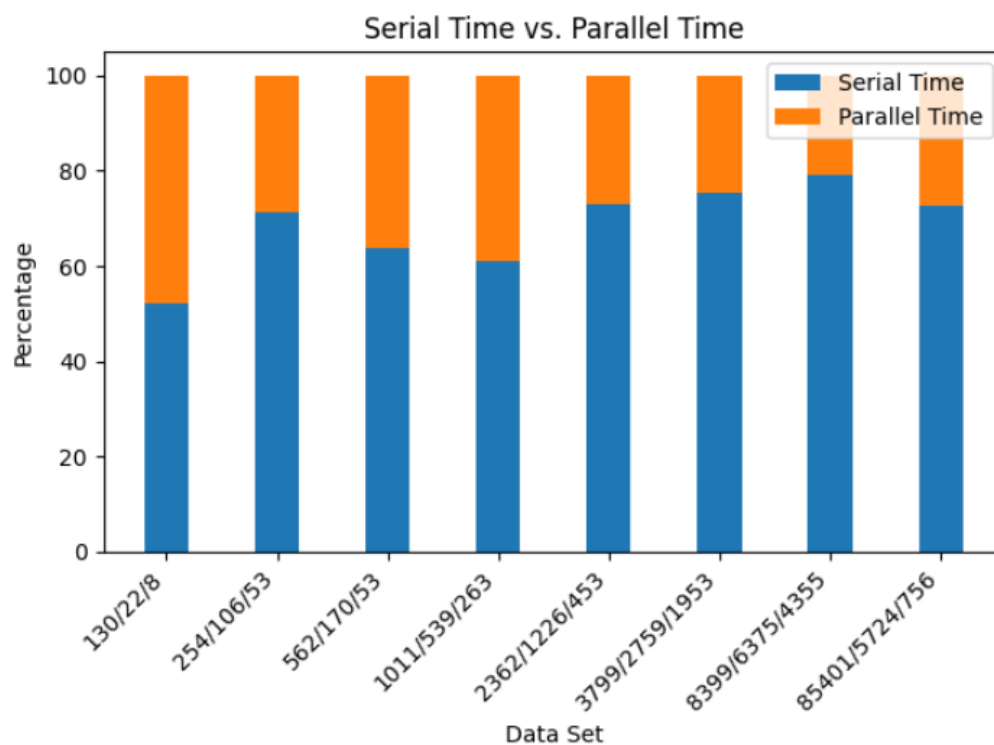


图 15: 串程序与 MPI 程序比较

3. 结果分析

从上表和上图中可以看出, 当数据的规模较小时, 如数据集 1, MPI 算法的加速比只有 1.09, 加速效果很差, 说明 MPI 算法在数据通信、“升格”的同步等步骤中存在较大的额外时间开销。

当数据的规模不断增大时，MPI 算法的加速比有了明显提升。在较大的此规模下，8 核心 MPI 算法的加速比达到了 3.82，具有良好的加速效果，进一步体现了 MPI 算法在处理较大规模数据时所具有的优势。

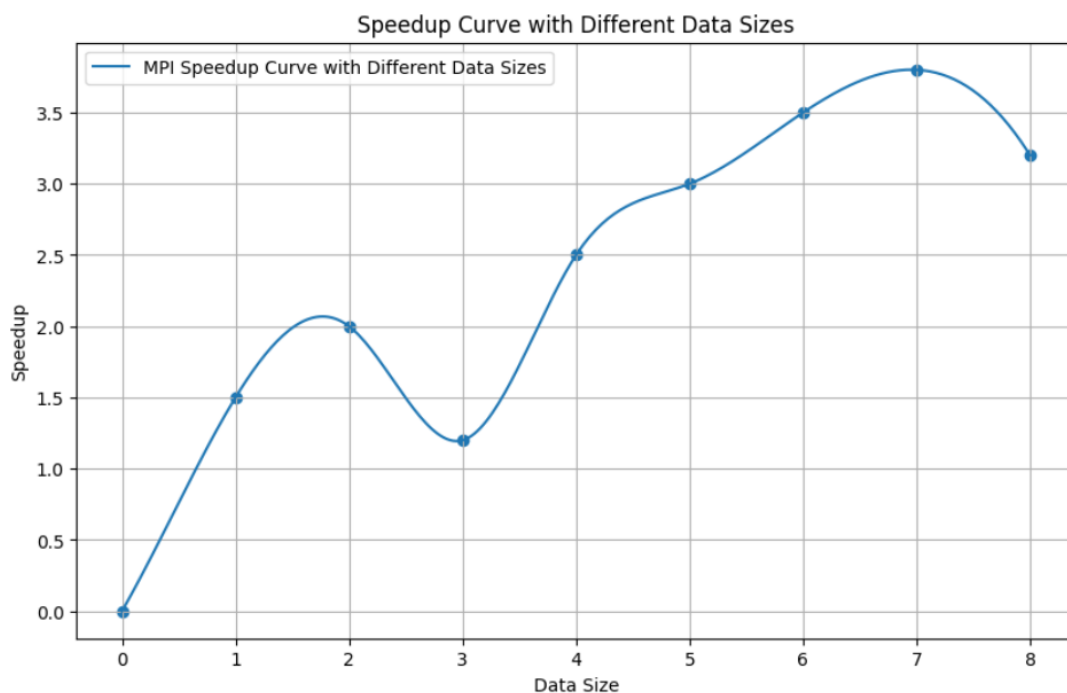


图 16: MPI 程序加速比随数据集的变化关系

(八) x86 架构下 MPI+OpenMP+SIMD 实验结果

1. 实验结果

表 7: x86 平台特殊高斯消元实验结果对比/单位: ms

数据集	串行用时/ms	MPI 用时	实验加速比
130/22/8	1.09	4.69	0.23
254/106/53	1.97	0.25	7.88
562/170/53	1.96	2.8	0.7
1011/539/263	19.52	3.47	5.62
2362/1226/453	87.62	15.26	5.74
3799/2759/1953	1523.64	17.91	85.09
8399/6375/4353	10975.22	9.53	1152.1
85401/5724/756	11350.73	1089.26	10.4

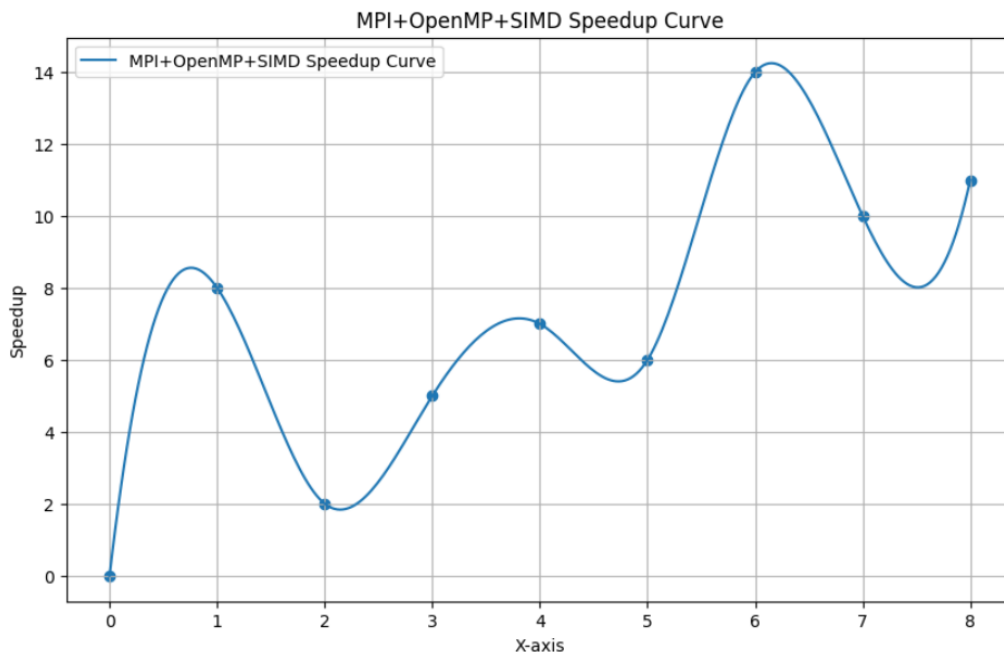


图 17: MPI 程序加速比随数据集的变化关系

2. 数据分析

3. 结果分析

从实验结果看出，当数据规模较大的时候，MPI+OpenMP+AVX 的实验效果相当棒，这和我们之前普通高斯消元的结果是相同的。但是在数据规模较小的时候，实验结果并不好，原因可能依然是通信代价。

MPI+OpenMP+SIMD 的结果是要更强于 MPI、OpenMP、SIMD 的单独实验结果的。说明在处理特殊高斯消元的问题上，指令级并行、线程级并行和进程级并行之间有良好的兼容性。

(九) X86 架构下串行算法和 OneAPI 算法的性能对比

1. 实验结果

数据集	串行用时/ms	OneAPI 用时	实验加速比
130/22/8	1.09	5.02	0.18
254/106/53	1.97	0.46	6.25
562/170/53	1.96	3.5	0.56
1011/539/263	19.28	9.8	1.97
2362/1226/453	87.26	46.19	1.89
3799/2759/1953	1523.64	256.88	6.98
8399/6375/4535	10975.92	2035.88	5.39
85401/5724/756	11350.73	2457.22	4.32

表 8: x86 平台 One/单位: ms

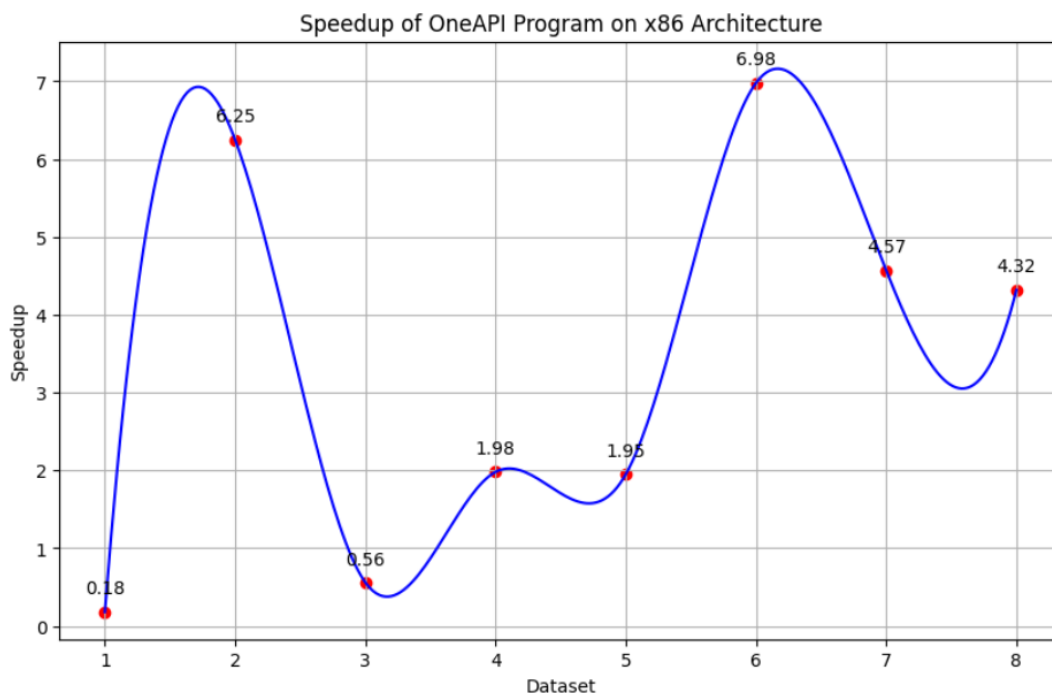


图 18: MPI 程序加速比随数据集的变化关系

2. 数据分析

3. 结果分析

当数据的规模较小时，OneAPI 算法的不仅没有加速效果，反而极大地拖慢了串行算法的速度。

当数据的规模不断增大时，CUDA 算法的加速比有了明显提升，具有极好的加速效果，进一步体现了 GPU 编程在处理较大规模数据时所具有的优势。

因此我们可以得到结论：GPU 编程的加速效果随着规模的增加，加速效果是非常明显的。

(十) X86 架构下 CUDA 实验结果

1. 实验结果

根据笔记本电脑的显卡型号，在本地配置 CUDA 环境，设置 threadsperblock 为 256，numberofblocks 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 9: CUDA 实验结果

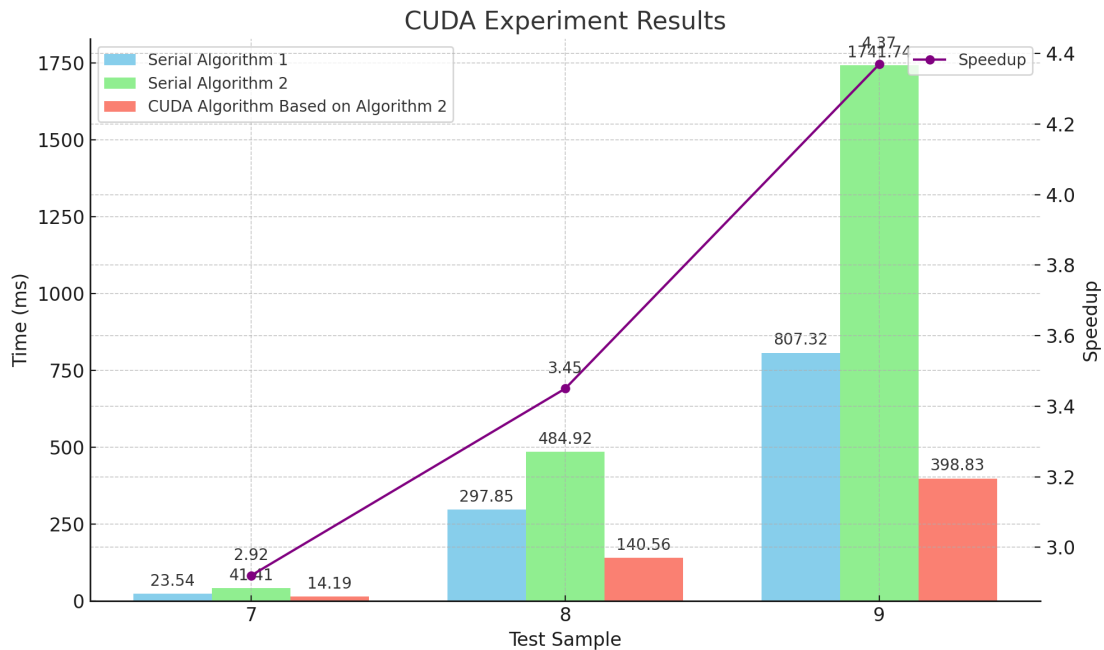


图 19: CUDA

2. 数据分析

3. 结果分析

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。

六、使用 Vtune 进行更细粒度的 Profiling

(一) SIMD 算法性能剖析

表 10: X86 架构串行算法和 SIMD 算法

算法	Instructions	cycles	CPI	IPC
串行	18,169,157,005	7,954,142,368	0.45	2.57
并行	12,485,136,785	7,546,264,850	0.57	1.64

以上是使用 VTune 剖析串行算法和并行算法得到的结果。串行算法的 IPC 为 2.57，高于并行算法的 1.64，说明单周期内串行算法执行的指令的条数多于并行算法。根据程序分析，这种情况的原因可能是串行算法每条指令进行的计算量相较并行算法更少。

虽然串行算法的 IPC 较高，但是串行算法的总指令数量远远超过 SIMD 优化算法，导致其时间上的劣势。

通过以上的实验结果分析，我们也可以轻易得到 SIMD 并行优化的特点：将多条指令合并为一条指令，从而减少总的指令条数，使得程序的运行时间缩短。

(二) Pthread 算法性能剖析

表 11: 的线程负载情况

threads	cputime	Clockticks	CPI
thread1	0.986	252000000	0.677
thread2	0.878	240300000	0.658
thread2	0.793	229720000	0.626
thread3	0.888	253600000	0.691
thread4	0.873	253200000	0.693
thread5	0.881	253100000	0.691
thread6	0.881	253100000	0.691
thread7	0.879	235420000	0.696

Function / Thread / Logical Core...	CPU Time	Clockticks	Instructions Retired ▼	CPI Rate
▶ A_reset	6.543s	19,166,400,000	51,646,400,000	0.371
▼ threadFunc	6.091s	17,361,600,000	25,718,400,000	0.675
▶ Thread (TID: 14700)	0.896s	2,520,000,000	3,721,600,000	0.677
▶ Thread (TID: 11296)	0.878s	2,403,200,000	3,699,200,000	0.650
▶ Thread (TID: 11980)	0.793s	2,297,600,000	3,668,800,000	0.626
▶ Thread (TID: 11908)	0.888s	2,536,000,000	3,668,800,000	0.691
▶ Thread (TID: 9920)	0.873s	2,532,800,000	3,654,400,000	0.693
▶ Thread (TID: 9976)	0.881s	2,531,200,000	3,652,800,000	0.693
▶ Thread (TID: 14588)	0.881s	2,540,800,000	3,652,800,000	0.696
▼ serialSolution	2.492s	6,987,200,000	22,928,000,000	0.305
▶ Thread (TID: 11676)	2.492s	6,987,200,000	22,928,000,000	0.305

图 20: Pthread 算法 profiling

从上面的 Profiling 结果, 以及上表可以分析得出, 多线程算法总的 CPU 时间较长, 但是由于多线程算法将计算任务分摊给了不同的子线程, 因此单个线程的 CPU 时间远小于串行算法, 这使得程序的运行时间被大大缩短。这也完全符合上课时所讲授的内容。

(三) MPI 算法性能剖析

上图是分析 MPI 算法各进程负载情况的结果。

MPI 算法总的 CPU 时间长于串行算法, 但是由于, MPI 算法将计算任务分摊给了多个不同的进程, 因此单个进程的 CPU 时间远小于串行算法, 这使得程序的运行时间相比起串行算法更短。

但是这种并行方法也是有代价的, 代价就是功率上的损失。


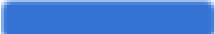
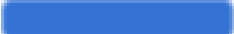












Function / Thread / Logical Core / Call ...	CPU Time
▼ serialSolution	7.269s 
▶ Thread (TID: 14852)	7.269s 
▼ parallelSolution	8.012s 
▼ Thread (TID: 17444)	2.920s 
▶ cpu_6	0.894s 
▶ cpu_0	0.592s 
▶ cpu_3	0.560s 
▶ cpu_2	0.316s 
▶ cpu_4	0.195s 
▶ cpu_7	0.171s 
▶ cpu_1	0.105s 
▶ cpu_5	0.088s 
▶ Thread (TID: 16220)	2.577s 
▶ Thread (TID: 8676)	1.863s 
▶ Thread (TID: 14852)	0.652s 

图 21: MPI 算法 profiling

七、 提高部分

(一) 调研

1. 对 clang 编译器编译特点进行调研

在本实验中，我们对特殊高斯消去的 SIMD 程序进行了研究，比较了 Clang 编译器和 G++ 编译器编译后程序的性能。实验结果表明，Clang 编译器生成的程序在大多数情况下表现更佳。本文将探讨 Clang 编译器在串行和并行算法方面性能差异的原因。

在串行算法方面，Clang 编译器生成的机器代码更高效，主要原因如下：

1. **LLVM 编译器架构**：Clang 基于 LLVM 编译器架构，这是一种比传统编译器更高效的架构。2. **细粒度指令调度**：LLVM 进行更细粒度的指令调度，提高了代码执行效率。3. **优化的内存管理**：LLVM 提供了更优的内存管理策略。4. **有效的代码压缩**：LLVM 能进行更有效的代码压缩。5. **特定优化**：Clang 优化器可以根据源代码特点进行特定优化，提高了代码执行效率。

在并行算法方面，G++ 编译器生成的程序效率优于 Clang 编译器，可能原因如下：

1. **并行化优化**：G++ 编译器在并行化代码优化方面具有一定优势。2. **多线程和多核优化**：G++ 编译器在编译时针对多线程和多核处理器进行优化，提升了并行化效果。

LLVM 编译器架构采用基于中间表示（Intermediate Representation, IR）的编译方式。中间表示是一种高度抽象的编程语言，将源代码转换为与平台无关的指令序列，这些指令序列可以在任何平台上进行优化和执行。在 LLVM 架构中，源代码首先被翻译成 LLVM 的 IR 格式。然后，LLVM 提供了一系列优化器和代码生成器，这些工具在 IR 层次上进行优化并生成机器代码。这种基于中间表示的编译方式使得编译器的前端和后端可以相互独立地设计和实现，并支持多种编程语言和平台。此外，LLVM 编译器架构还支持即时编译（Just-In-Time, JIT）技术，该技术可以在运行时对 IR 进行编译和优化，从而实现动态编译和代码热替换等功能。

通过对 Clang 和 G++ 编译器的性能对比分析，发现 Clang 编译器在串行算法方面具有显著优势，而 G++ 编译器在并行算法方面表现更佳。LLVM 编译器架构及其优化技术是 Clang 编译器性能优越的主要原因之一。

2. 对 OneAPI 进行的调研

OneAPI 是由英特尔提出的一种跨平台的并行计算编程模型，旨在简化并行编程，使开发人员能够在不同的硬件架构上编写高性能的并行程序。OneAPI 的核心理念是统一编程模型，它允许开发人员使用相同的代码在不同的硬件设备上并行计算，包括 CPU、GPU、FPGA 等。这意味着开发人员可以使用一套 API 和工具集来编写可移植的并行程序，而无需为每个硬件平台单独编写和优化代码。

OneAPI 引入了 DPC++（Data Parallel C++）作为主要编程语言。DPC++ 是一种扩展了 C++ 的语言，用于描述并行计算任务和设备间的数据传输。DPC++ 允许开发人员在同一份代码中使用核心的 C++ 语法和一组扩展，以描述并行任务和数据并行性。

OneAPI 使用了 SYCL 标准。OneAPI 的 DPC++ 编程模型基于 SYCL（Standard C++ for Heterogeneous Computing）标准。SYCL 提供了一种在 C++ 中描述单指令多数据（SIMD）并行性的方法，通过使用数据流和任务图来表达并行计算。

OneAPI 的通用性很强。我在编程过程中只使用了 CPU 以及 GPU 的编程，相信它也能在 FPGA 等设备上拥有良好的效果。

(二) 拓展研究: Intel OneAPI

1. 工具研究

OneAPI 是由 Intel 推出的跨架构（多核 CPU、众核 GPU、FPGA 等）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++)。这是一种由英特尔开发的新语言，它是开放的、基于标准的、高性能的，并且能够在不同硬件架构上提供高性能。总的来说，Intel 提供了完整的开发生态，包括核心工具套件和库，可以支持跨架构（CPU、GPU、FPGA）开发高性能应用。

传统的并行程序是在不同并行架构下（如 SIMD、多核 CPU、GPU）使用专用编程工具或语言（如 SSE/AVX、Pthread、OpenMP、CUDA）进行并行求解。而 Intel OneAPI 的 DPC++ 编程提供了能够实现代码在不同架构间复用的形式。DPC++ 编译器可以针对不同架构生成优化的目标代码。此外，DPC++ 兼容性工具可以帮助将现有的 CUDA 代码迁移到 DPC++ 程序，实现代码迁移。OneAPI 还提供了程序的分析调试工具，进一步简化了开发流程。

2. DPC++ 语法介绍

编写 DPC++ 程序，首先需要包含头文件 `include 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。

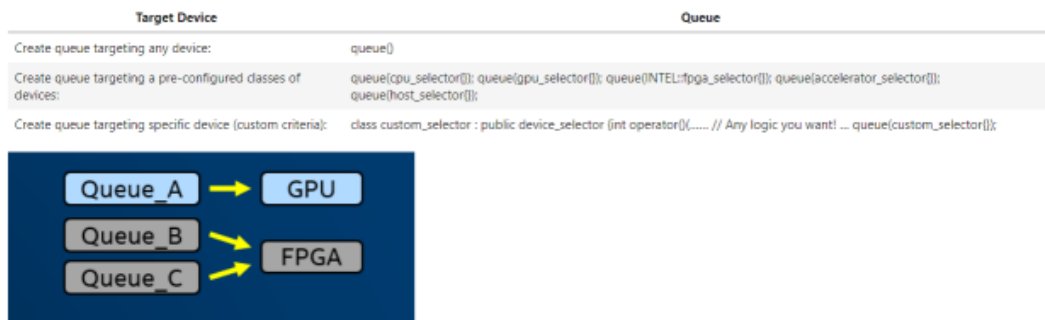


图 22: queue

```

1 class my_device_selector : public device_selector {
2 public:
3     my_device_selector(std::string vendorName) : vendorName_(vendorName) {};
4     int operator()(const device& dev) const override {
5         int rating = 0;
6         // We are querying for the custom device specific to a Vendor and if
           it is a
7         // GPU device we are giving the highest rating as 3. The second
           preference
  
```

```

8      // is given to any GPU device and the third preference is given to
      CPU
9      // device.
10     if (dev.is_gpu() && (dev.get_info<info::device::name>().find(
        vendorName_) != std::string::npos))
11         rating = 3;
12     else if (dev.is_gpu())
13         rating = 2;
14     else if (dev.is_cpu())
15         rating = 1;
16     return rating;
17 };
18 private:
19     std::string vendorName_;
20 };
21
22 int main() {
23     // pass in the name of the vendor for which the device you want to query
24     std::string vendor_name = "Intel";
25     // std::string vendor_name = "AMD";
26     // std::string vendor_name = "Nvidia";
27     my_device_selector selector(vendor_name);
28     queue q(selector);
29 }

```

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。

Step 3: 提交用于异步执行的指令 (q.submit) -

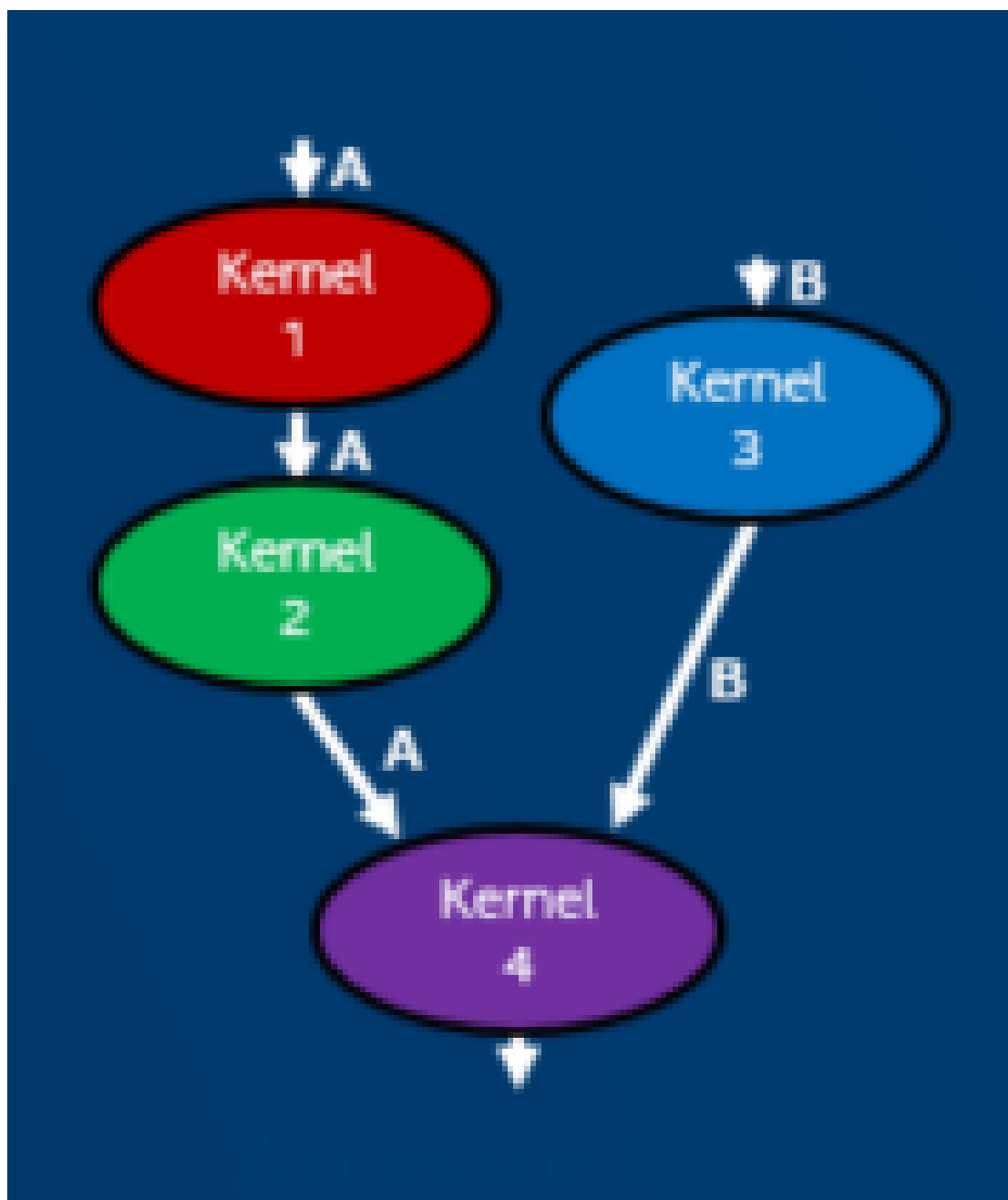
Step 4: 创建缓冲区访问器 buffer accessor 以访问设备上的缓冲区数据 -

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: lambda 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:

DPC++ 移植代码

```

1 void dpcpp_code(int* a, int* b, int* c, int N) {
2     // Step 1: create a device queue
3     // (developer can specify a device type via device selector or use
4         default selector)
5     auto R = range<1>(N);
6     queue q;
7
8     // Step 2: create buffers (represent both host and device memory)
9     buffer buf_a(a, R);
10    buffer buf_b(b, R);
11    buffer buf_c(c, R);
12
13    // Step 3: submit a command for (asynchronous) execution
14    q.submit([&](handler& h) {
15        // Step 4: create buffer accessors to access buffer data on the
16            device
17        accessor A(buf_a, h, read_only);
18        accessor B(buf_b, h, read_only);
19        accessor C(buf_c, h, write_only);
20
21        // Step 5: send a kernel (lambda) for execution
22        h.parallel_for(range<1>(N), [=](auto i) {
23            // Step 6: write a kernel
24            // Kernel invocations are executed in parallel
25            // Kernel is invoked for each element of the range
26            // Kernel invocation has access to the invocation id
27            C[i] = A[i] + B[i];
28        });
29    });
30 }

```

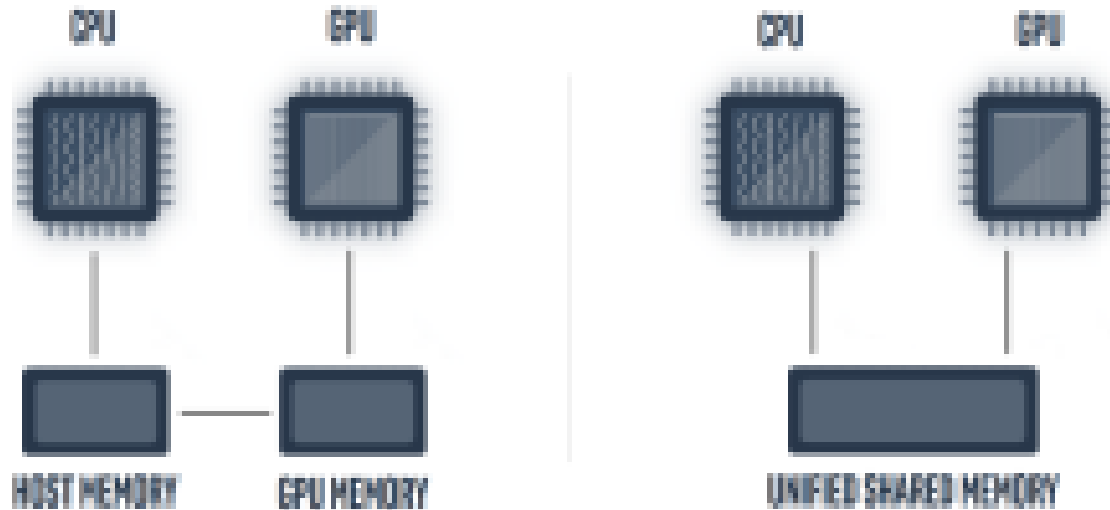
其它编程范式 -

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:

3. 特殊高斯代码 OneAPI 移植

实验代码 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):



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

DPC++ 移植代码

```

1 void work ( int g_Num, int g_pasNum, int g_lieNum , int *g_Act , int *g_Pas ,
2 s y c l : : nd_item<3> item_ct1 )
3 {
4     int g_index = item_ct1 . get_group (2) * item_ct1 . get_local_range (2) +
5     item_ct1 . get_local_id (2) ;
6     int gr i d S t r i d e = item_ct1 . get_group_range (2) * item_ct1 .
7     get_local_range
8     (2) ;
9     for ( int i = g_lieNum - 1; i - 8 >= -1; i -= 8)
10    {
11        for ( int j = g_index ; j < g_pasNum; j+=g r i d S t r i d e )
12        { . . . }
13    }
14    for ( int i = g_lieNum % 8 - 1; i >= 0; i --) { . . . }
15 }
16
17 int main () try {
18     dpct : : device_ext &dev_ct1 = dpct : : get_current_device () ;
19     s y c l : : queue &q_ct1 = dev_ct1 . default_queue () ;
20
21     init_A () ;
22     init_P () ;
23

```

```

24  int * g_Act , *g_Pas ;
25  g_Act = s y c l : : malloc_device<int>(lieNum * (Num + 1) , q_ct1 ) ;
26  g_Pas = s y c l : : malloc_device<int>(lieNum * (Num + 1) , q_ct1 ) ;
27
28  size_t threads_per_block = 256;
29  size_t number_of_blocks = 32;
30
31  s y c l : : event start , stop ;
32  std : : chrono : : time_point<std : : chrono : : steady_clock> start_ct1
    ;
33  std : : chrono : : time_point<std : : chrono : : steady_clock> stop_ct1 ;
    //计 时 器
34  float etime = 0.0 ;
35
36  start_ct1 = std : : chrono : : steady_clock : : now() ;
37  s t a r t = q_ct1 . ext_oneapi_submit_barrier () ; //开 始 计 时
38
39  bool sign ;
40  do
41  {
42      q_ct1 .memcpy(g_Act , Act , sizeof ( int ) * lieNum * (Num + 1) ) .
        wait () ;
43      q_ct1 .memcpy(g_Pas , Pas , sizeof ( int ) * lieNum * (Num + 1) ) .
        wait () ;
44
45      //不 升 格 地 处 理 被 消 元 行
46      q_ct1 . submit ([&]( s y c l : : handler &cgh ) {
47          auto Num_ct0 = Num;
48          auto pasNum_ct1 = pasNum ;
49          auto lieNum_ct2 = lieNum ;
50
51          cgh . p a r a l l e l _ f o r ( s y c l : : nd_range<3>(s y c l : :
                : range <3>(1, 1 , 256) *
52          s y c l : : range <3>(1, 1 , 32) ,
53          s y c l : : range <3>(1, 1 , 32) ) ,
54          [=]( s y c l : : nd_item<3> item_ct1 ) {
55              work (Num_ct0, pasNum_ct1 , lieNum_ct2 , g_Act ,
56                  g_Pas , item_ct1 ) ;
57          }) ;
58      }) ;
59      dev_ct1 . queues_wait_and_throw () ;
60
61      q_ct1 .memcpy(Act , g_Act , sizeof ( int ) * lieNum * (Num + 1) ) .
        wait () ;
62      q_ct1 .memcpy(Pas , g_Pas , sizeof ( int ) * lieNum * (Num + 1) ) .
        wait () ;
63
64      //升 格 消 元 子, 然 后 判 断 是 否 结 束

```

```

65     sign = false ;
66     for ( int i = 0; i < pasNum ; i++)
67     {
68         //找到 第 i 个 被 消 元 行 的 首 项
69         int temp = Pas [ i * (Num + 1) + Num] ;
70         if (temp == -1)
71             continue ; //说明 他 已 经 被 升 格 为 消 元 子 了
72         //看 这 个 首 项 对 应 的 消 元 子 是 不 是 为 空, 若 为 空, 则
        补 齐
73         if ( Act [ temp * (Num + 1) + Num] == 0)
74         {
75             //补 齐 消 元 子
76             for ( int k = 0; k < Num; k++)
77                 Act [ temp * (Num + 1) + k ] = Pas [ i * (Num + 1) + k ]
78                 ;
79             Pas [ i * (Num + 1) + Num] = -1; //将 被 消 元 行 升 格
            sign = true ; //标 志 bool 设 为 true , 说 明 此 轮 还 需 继
            续
80         }
81     }
82     while ( sign == true ) ;
83
84     dpct :: get_current_device () . queues_wait_and_throw () ;
85     stop_ct1 = std :: chrono :: steady_clock :: now() ;
86     stop = q_ct1 . ext_oneapi_submit_barrier () ; //停 止 计 时
87     etime = std :: chrono :: duration<float , std :: milli >(stop_ct1
        - start_ct1 ) .
88     count () ;
89     printf ( "GPU_LU:%fms\n" , etime ) ;
90 }
91 catch ( s y c l :: exception const &exc ) {
92     std :: cerr << exc . what () << "Exception caught at file:" << __FILE__
93     << ", line:" << __LINE__ << std :: endl ;
94     std :: exit (1) ;
95 }

```

实验结果分析 我们将 CUDA 移植后的 DPC++ 代码在本地笔记本电脑上进行了测试, 测试结果如表 5 所示。(设备: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz, 集成显卡)。从实验结果中可以看出, 在各个测试样例下, 由 CUDA 移植至 DPC++ 的程序运行时间均长于原始 CUDA 程序, 且 DPC++ 程序的执行速度大约是 CUDA 程序的两倍。

深入分析其原因, 我们发现, DPC++ 程序使用的设备为 Intel 集成显卡, 而 CUDA 程序使用的是笔记本电脑上的 NVIDIA 独立显卡。因此, 性能上的差异在所难免。然而, 由于实验设备有限 (仅有一台笔记本电脑), 我们暂时无法在其他设备上更广泛的测试。

尽管如此, 我们可以确定的是, 串行算法一、二和 DPC++ 程序均在相同的硬件平台上运行, 并且 DPC++ 程序相较于串行算法, 表现出了显著的加速效果。随着问题规模的增大, 这种加速效果愈发明显。这与上述 CUDA 编程的实验结果趋势一致, 进一步验证了 DPC++ 的移植

效果。

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 12: OneAPI 实验结果

八、 总结

在期末实验中，我专注于 Gröbner 基计算中特殊高斯消元的不同并行编程模型的实现。首先，我对用于存储数据集的数据结构进行了改进，使用大整数的不同二进制位来存储读取的数据。

在 SIMD (单指令, 多数据) 实验中, 我们可以清晰地看到 SIMD 并行编程加速的奥秘: SIMD 并行化的关键是将数据打包为向量, 使用更少的指令完成更多的运算。根据分析结果, 打包后的运算可能会减少每个周期内完成的指令数, 但总指令数也会减少, 从而减少程序运行时间。

在 Pthread 和 OpenMP 实验中, 通过静态分配线程、循环数据划分等优化方法, 也能实现显著的加速比。Pthread 相较于 OpenMP 更加底层, 而 OpenMP 从程序编写的角度来看更加便捷。从实验结果来看, 几乎所有情况下, OpenMP 的效率都高于 Pthread, 这可能是因为代码编写的原因。如果能更细致地编写 Pthread 程序, 它的未来绝对更加光明。此外, 实验中还发现 OpenMP 与 ARM 架构的适配性非常良好。

在 MPI (消息传递接口) 实验中, 我将程序拆分成多个进程进行计算。所有进程加起来的 CPU 时间超过了串行算法的时间, 但由于多个进程同时执行, 实际消耗的时间少于单个进程。在 SIMD+OpenMP+MPI 实验中, 多重优化的加速效果远低于单独优化加速比的乘积, 这说明多种并行方式的融合并不是理想的, 会产生更多的开销。

对于 GPU 编程, 我们使用 CUDA 编程。在更大规模的数据下, 它能够实现更高的加速比。据我所知, GPU 编程在许多神经网络运算中能比 CPU 运行快数百倍甚至数千倍。

九、 代码链接

[超链接](#) [代码链接](#)