南開大學
Nankai University

计算机学院

并行程序设计实验报告

Pthread 加速的高斯消去算法

2023 年 6 月 2 日

目录

1 实验目标

1. 通过 Pthread, 在 X86 和 ARM 平台上实现多线程加速的高斯消去算法和消元子模式的高斯消去算法。
2. 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量的加速比。
3. 利用 perf 等工具对实验结果进行分析，找出性能差异的原因。

2 核心代码

2.1 普通高斯消去算法

2.1.1 串行算法

串行算法的实现和之前一样，不做更改，代码如下：

```
#define N 1024
#define ele_t float

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

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

2.1.2 Pthread 并行（重复创建线程）

观察串行算法的代码，我们可以发现，高斯消去算法中包含有三重循环：第一重循环遍历消元行、第二重循环遍历被消元行、第三重循环将被消元行减去消元行。在进行 SIMD 并行化时，我们的工作主要集中在第三重循环上，但当我们来到 Pthread 并行时，如果在第三重循环上进行并行，并且矩阵规模不大时，程序花在线程间通讯的时间代价将是不可接受的，因此，并行化的工作应该着眼于第二重循环上。而这一重循环上也无须考虑任何数据依赖。

决定好并行化的循环之后，作者的并行化思路如下：将需要消去的行数平均分给各个线程（每个线程分到 $(n-i-1)/NUM\_THREADS$ 行），并将指向矩阵的指针和当前消去行、开始被消去行、

需要被消去的行数传递给线程函数。之后主线程再计算无法被整除的几行。具体的子线程和参数结构体代码如下：

```
struct LU_data
{
    pthread_mutex_t finished = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t startNext = PTHREAD_MUTEX_INITIALIZER;
    ele_t (*mat)[N][N];
    int n; // 矩阵行数/列数
    int i, begin, nLines; // 当前行、开始消去行、需要消去的行数
};

void *subthread_LU(void *_params)
{
    LU_data *params = (LU_data *)_params;
    int i = params->i, n = params->n;
    float32x4_t mat_j, mat_i, div4;
    for (int j = params->begin; j < params->begin + params->nLines; j++)
    { // 遍历列
        if ((*params->mat)[i][i] == 0) // 不需要消去
            continue;
        ele_t div = (*params->mat)[j][i] / (*params->mat)[i][i];
        div4 = vmovq_n_f32(div);
        for (int k = i / 4 * 4; k < n; k += 4)
        {
            mat_j = vld1q_f32((*params->mat)[j] + k);
            mat_i = vld1q_f32((*params->mat)[i] + k);
            vst1q_f32((*params->mat)[j] + k, vmlsq_f32(mat_j, div4, mat_i));
        }
    }
}
```

用于管理线程、分配任务的函数代码如下。由于作者上手 Pthread 初期，对创建线程消耗的认知还不是很清晰，因此采用了在第二重循环中，每个循环步都会创建 $NUM\_THREADS$ 个线程。这样的策略导致创建线程的次数为 $n * NUM\_THREADS$ 次，造成了极大的性能浪费。即便后来引入当每个线程能分配到的行数小于 31 行时便不启用并行处理的机制，并行算法的加速比也很难超过 1.5。

```c
void LU_pthread(ele_t mat[N][N], int n)
{
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    pthread_t threads[NUM_THREADS];
    LU_data attr[NUM_THREADS];

    for (int i = 0; i < n; i++)
    { // 遍历行
        int nLines = (n - i - 1) / NUM_THREADS; // 每个线程需要处理的矩阵行数
        if (nLines > 31)
        { // 数据比较大，才有多线程的必要
            for (int th = 0; th < NUM_THREADS; th++)
            {
                attr[th].th = th;
                attr[th].mat = &new_mat;
                attr[th].n = n;
                attr[th].i = i;
                attr[th].nLines = nLines;
                attr[th].begin = i + 1 + th * nLines;
                int err = pthread_create(&threads[th], NULL, \
                                subthread_LU, (void *)&attr[th]);
                // 省略：创建线程失败，退出程序
            }
            // 算掉无法被整除的最后几行
            for (int j = i + 1 + NUM_THREADS * ((n - i - 1) / NUM_THREADS); j < n; j++)
            {
                if (new_mat[i][i] == 0)
                    continue;
                ele_t div = new_mat[j][i] / new_mat[i][i];
                for (int k = i; k < n; k++)
                    new_mat[j][k] -= new_mat[i][k] * div;
            }

            for (int th = 0; th < NUM_THREADS; th++)
                pthread_join(threads[th], NULL); //保证所有线程完成任务
        }
        else
            // 省略：如果 nLines 比较小，则进行串行处理
    }
}
```

2.1.3 Pthread 并行（线程只创建一次）

为了防止重复创建线程，作者使用了两个互斥锁实现了对线程的反复利用。在线程参数结构体中包含了 startNext 和 finished 互斥锁，在真正的运算开始前，会将所有线程都创建出来，并且创建前将两个互斥锁都上锁；在需要用到线程进行运算时，则将线程参数结构体中的数据赋值，然后把 startNext 解锁，这时各个线程就会开始进行运算；运算完毕后，线程会把 finished 解锁，这时主线程就可以通过 pthread_mutex_lock(finished) 来实现类似 pthread_join 的效果。

```
void *subthread_static_LU(void *_params)
{
    LU_data *params = (LU_data *)_params;
    float32x4_t mat_j, mat_i, div4;
    while (true)
    {
        pthread_mutex_lock(&(params->startNext));
        int i = params->i, n = params->n; // 一定要等解锁后再读数据!!!
        // 省略：进行运算
        pthread_mutex_unlock(&(params->finished)); // 做完了
    }
}

void LU_static_thread(ele_t mat[N][N], int n)
{
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    pthread_t threads[NUM_THREADS];
    LU_data attr[NUM_THREADS];
    for (int th = 0; th < NUM_THREADS; th++)
    {
        pthread_mutex_lock(&(attr[th].startNext));
        pthread_mutex_lock(&(attr[th].finished));
        pthread_create(&threads[th], NULL, subthread_static_LU, (void *)&attr[th]);
        // 省略：线程创建错误处理
    }
    for (int i = 0; i < n; i++)
    {
        int nLines = (n - i - 1) / NUM_THREADS;
        for (int th = 0; th < NUM_THREADS; th++)
        {
            attr[th].th = th;
            // ... 省略：给其他 attr 中的参数赋值...
            pthread_mutex_unlock(&(attr[th].startNext));
        }
```

```
35
36          for (int j = i + 1 + NUM_THREADS * ((n - i - 1) / NUM_THREADS); j < n; j++)
37              // 省略：算掉无法被整除的最后几行
38
39          for (int th = 0; th < NUM_THREADS; th++)
40              pthread_mutex_lock(&(attr[th].finished)); // 等待所有线程完成
41      }
42  }
```

2.2 消元子模式的高斯消去算法

2.2.1 数据结构相关代码

本次实验的特殊高斯消去算法中，我们使用了与上次实验相同的数据结构，也就是被消元行按位图格式存储，消元子按位图格式扩展为正方形矩阵，并将对应 i 列的消元子放在 i 行中。因此数据结构相关的代码省略。

2.2.2 新串行算法

上一次特殊高斯消去实验中，我们使用了如下的算法：

```
1   void groebner(mat_t ele[COL][COL / mat_L + 1], mat_t row[ROW][COL / mat_L + 1])
2   { // ele= 消元子，row= 被消元行
3       memcpy(ele_tmp, ele, sizeof(mat_t) * COL * (COL / mat_L + 1));
4       memcpy(row_tmp, row, sizeof(mat_t) * ROW * (COL / mat_L + 1));
5       for (int i = 0; i < ROW; i++)
6       { // 遍历被消元行
7           for (int j = COL; j >= 0; j--)
8           { // 遍历列
9               if (row_tmp[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
10              { // 当前位置有元素，需要消元
11                  if (ele_tmp[j][j / mat_L] & ((mat_t)1 << (j % mat_L)))
12                  { //找到对应消元子
13                      for (int p = COL / mat_L; p >= 0; p--)
14                          row_tmp[i][p] ^= ele_tmp[j][p];
15                  }
16                  else
17                  { // 找不到对应消元子，升格当前消元行
18                      memcpy(ele_tmp[j], row_tmp[i], (COL / mat_L + 1) * sizeof(mat_t));
19                      break;
20                  }
21              }
22          }
```

```
    }
    // 省略：以稀疏矩阵格式输出 row_tmp，已与数据集中的参考结果对比一致
}
```

不难注意到，这一算法将被消元行作为了最外层循环遍历的对象，而第二层循环遍历的对象则是当前被消元行的每一个元素，如果当前元素为 1，需要进行消元，且找不到对应消元子，就将当前被消元行升格为消元子。这就导致如果我们的并行化如果要在第二层，就不得不考虑消元顺序的问题，这使得在第二层的并行化几乎不太可能实现。因此，作者设计了如下的新串行算法。

```
void groebner_new(mat_t ele[COL][COL / mat_L + 1], mat_t row[ROW][COL / mat_L + 1])
{ // ele= 消元子，row= 被消元行
    memcpy(ele_tmp, ele, sizeof(mat_t) * COL * (COL / mat_L + 1));
    memcpy(row_tmp, row, sizeof(mat_t) * ROW * (COL / mat_L + 1));
    bool upgraded[ROW] = {0}; // 被消元行已升格标识
    for (int j = COL; j >= 0; j--)
    { // 遍历消元子
        if (ele_tmp[j][j / mat_L] & ((mat_t)1 << (j % mat_L)))
        { // 如果存在对应消元子则进行消元
            for (int i = 0; i < ROW; i++)
            { // 遍历被消元行
                if (upgraded[i])
                    continue;
                if (row_tmp[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
                { // 如果当前行需要消元
                    for (int p = COL / mat_L; p >= 0; p--)
                        row_tmp[i][p] ^= ele_tmp[j][p];
                }
            }
        }
        else
        { // 不存在对应消元子，则找出第一个被消元行升格
            for (int i = 0; i < ROW; i++)
            { // 遍历被消元行
                if (upgraded[i])
                    continue;
                if (row_tmp[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
                {
                    memcpy(ele_tmp[j], row_tmp[i], (COL / mat_L + 1) * sizeof(mat_t));
                    upgraded[i] = true;
                    j++;
                    break;
                }
```

```
            }
        }
    }
}
```

在这个算法中，最外层循环遍历的是消元子矩阵（正方形矩阵的行）。如果在第 j 行有对应消元子（元素 (j,j) 为 1），则遍历所有被消元行，如果其第 j 列有值，则进行消去操作；而如果第 j 行没有对应消元子，则遍历被消元行，并将第一个 j 列有值的被消元行升格为消元子。这样，第二层循环上遍历被消元行的操作便不会有数据依赖关系，可以轻松的并行化。

2.2.3 Pthread 并行

有了如上的新串行算法，进行 Pthread 并行就非常简单了。消元子模式的高斯消去算法的并行化思路和普通高斯消去算法的并行化思路非常相似，都是通过 $startNext$ 和 $finished$ 两个互斥锁来实现线程通信和同步，因此这里不再赘述了。

```
struct groebnerData
{
    mat_t (*ele)[COL][COL / mat_L + 1];
    mat_t (*row)[ROW][COL / mat_L + 1];
    bool (*upgraded)[ROW];
    int j, begin, nLines;
    pthread_mutex_t finished = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t startNext = PTHREAD_MUTEX_INITIALIZER;
};

void *subthread_groebner(void *_params)
{
    groebnerData *params = (groebnerData *)_params;
    int j;
    simd_reg_t row_i, ele_j;
    while (true)
    {
        pthread_mutex_lock(&(params->startNext));
        j = params->j;
        for (int i = params->begin; i < params->begin + params->nLines; i++)
        { // 遍历被消元行
            if ((*params->upgraded)[i])
                continue;
            if ((*params->row)[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
            { // 如果当前行需要消元
                for (int p = COL / mat_L; p >= 0; p--)
                    (*params->row)[i][p] ^= (*params->ele)[j][p];
```

```
28                  // SIMD 代码省略
29          pthread_mutex_unlock(&(params->finished));
30      }
31  }
32
33  void groebner_pthread(mat_t ele[COL][COL / mat_L + 1], mat_t row[ROW][COL / mat_L + 1])
34  {
35      // ele= 消元子, row= 被消元行
36      memcpy(ele_tmp, ele, sizeof(mat_t) * COL * (COL / mat_L + 1));
37      memcpy(row_tmp, row, sizeof(mat_t) * ROW * (COL / mat_L + 1));
38
39      bool upgraded[ROW] = {0};
40      pthread_t threads[NUM_THREADS];
41      groebnerData attr[NUM_THREADS];
42
43      for (int th = 0; th < NUM_THREADS; th++)
44      {
45          pthread_mutex_lock(&(attr[th].startNext));
46          pthread_mutex_lock(&(attr[th].finished));
47          pthread_create(&threads[th], NULL, subthread_groebner, (void *)&attr[th]);
48      } // 省略: 创建线程失败处理
49
50      for (int j = COL; j >= 0; j--)
51      { // 遍历消元子
52          if (ele_tmp[j][j / mat_L] & ((mat_t)1 << (j % mat_L)))
53          { // 如果存在对应消元子则进行消元
54              int nLines = ROW / NUM_THREADS;
55              for (int th = 0; th < NUM_THREADS; th++)
56              {
57                  attr[th].ele = &ele_tmp;
58                  // 省略: 给其他属性赋值
59                  pthread_mutex_unlock(&(attr[th].startNext));
60              }
61
62              for (int i = ROW / NUM_THREADS * NUM_THREADS; i < ROW; i++)
63              { // 遍历被消元行
64                  if (upgraded[i]) //已经升格
65                      continue;
66                  if (row_tmp[i][j / mat_L] & ((mat_t)1 << (j % mat_L)))
67                  { // 如果当前行需要消元
68                      for (int p = COL / mat_L; p >= 0; p--)
69                          row_tmp[i][p] ^= ele_tmp[j][p];
```

```
                }
            }

            for (int th = 0; th < NUM_THREADS; th++)
                pthread_mutex_lock(&(attr[th].finished)); // 等待线程处理完
        }
        else
            ;// 省略：不存在对应消元子，则找出第一个被消元行升；格
    }
}
```

3 实验环境

3.1 X86 平台

X86 平台的实验使用了英特尔的 DevCloud 平台。服务器操作系统为 Ubuntu 20.04.4 LTS，Kernel 版本 5.4.0-80-generic；编译器为 gcc，版本 9.4.0；CPU 型号为 Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz，内存 188Gi，虚拟化方式为 VT-x；CPU Cache 等信息如下 lscpu 命令输出所示：

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          24
On-line CPU(s) list:             0-23
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           85
Model name:                      Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Stepping:                        4
CPU MHz:                         1200.428
CPU max MHz:                     3700.0000
CPU min MHz:                     1200.0000
BogoMIPS:                        6800.00
Virtualization:                  VT-x
L1d cache:                       384 KiB
L1i cache:                       384 KiB
L2 cache:                        12 MiB
```

```
L3 cache:                        38.5 MiB
NUMA node0 CPU(s):               0-5,12-17
NUMA node1 CPU(s):               6-11,18-23
```

3.2 ARM 平台

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

4 实验过程

4.1 普通高斯消去

首先生成测试数据，之后运行测试脚本。

```
g++ ./datagen.cpp -o datagen
./datagen
qsub sub_gauss.sh # 鲲鹏云服务器
bash gauss_timing.sh #DevCloud
```

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

4.2 消元子模式的高斯消去

对于 X86 上的实验，需要将 Groebner 数据集拷贝至脚本所在目录的上级目录下。

```
qsub subgroebner.sh # 鲲鹏云服务器
bash groebner_timing.sh #DevCloud
```

这里使用的脚本可以在GitHub 仓库中找到，脚本中的编译命令使用的编译器均为 gcc，均使用了-pthread 选项、-O0 选项和-w 选项。

5 实验结果

5.1 普通高斯消去

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

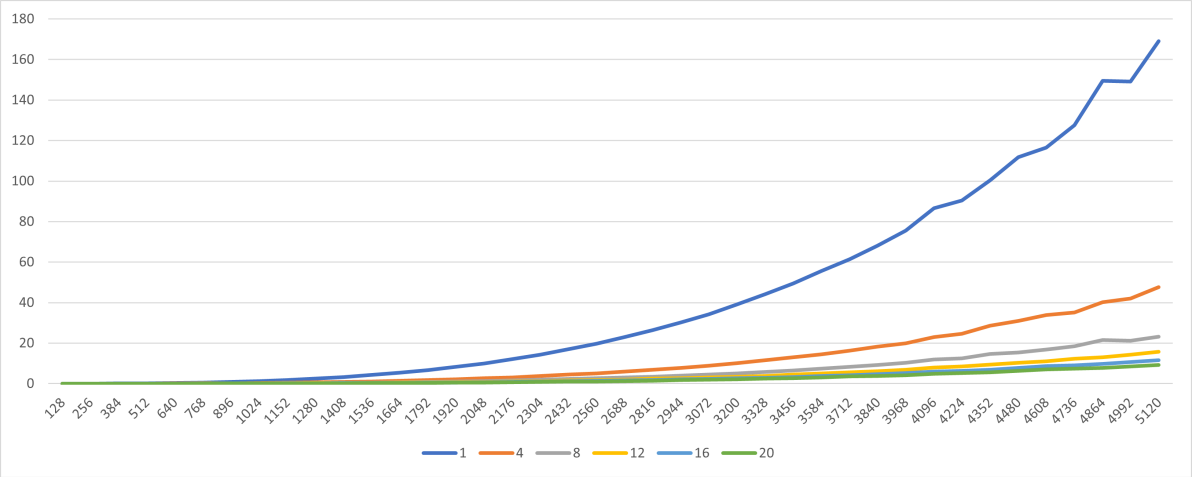

图 5.1: ARM 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

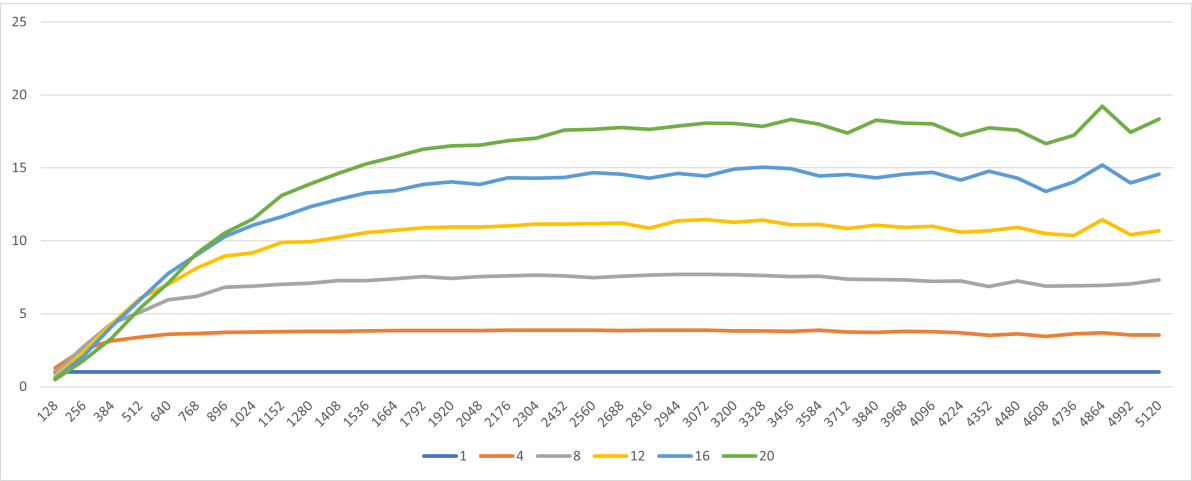

图 5.2: ARM 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

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

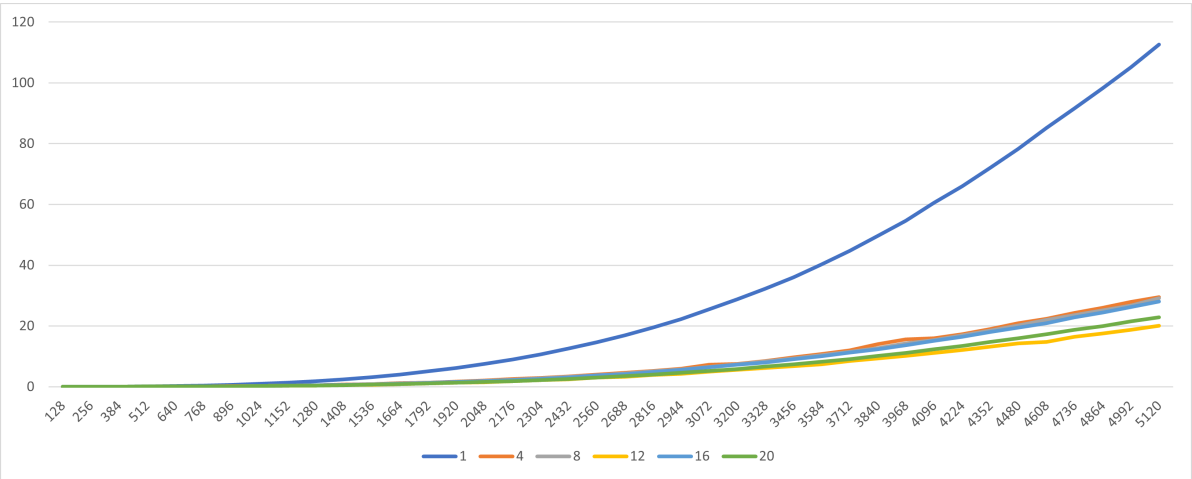

图 5.3: X86 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

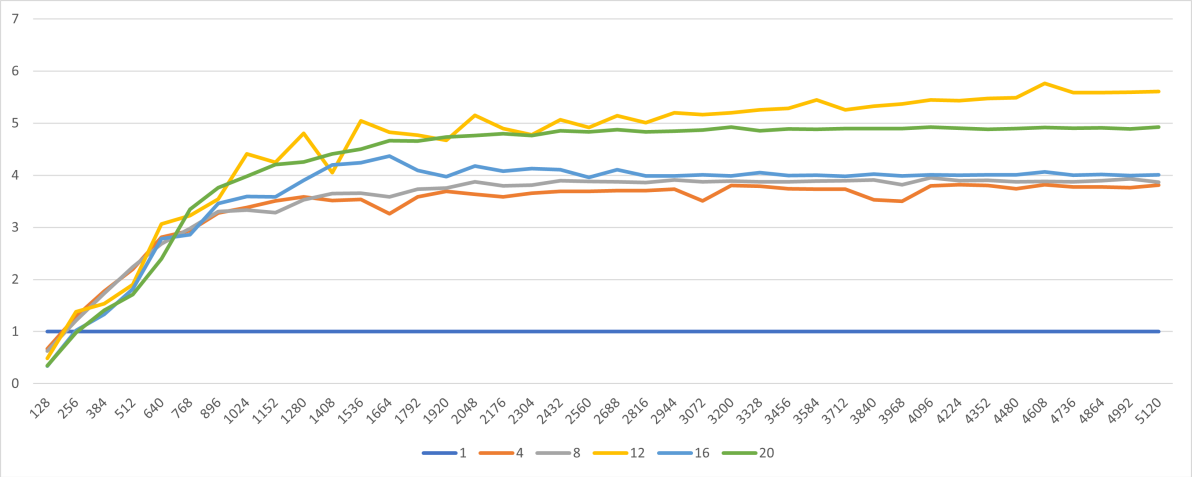

图 5.4: X86 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

5.2 消元子模式的高斯消去

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

矩阵列数 \ 线程数	1	4	8	12	16	20
130	6.20E-06	0.000657981	0.00151675	0.00180888	0.00214525	0.00263356
254	0.000203302	0.00264929	0.0041691	0.00517378	0.00733332	0.00892373
562	0.000545487	0.00422865	0.00666771	0.00898492	0.00953388	0.0146113
1011	0.00954177	0.0148444	0.0204453	0.0250353	0.028699	0.0353505
2362	0.0423403	0.0616395	0.0698515	0.070898	0.0820665	0.0889781
3799	0.692533	0.36458	0.346752	0.39223	0.487133	0.488261
8399	9.08002	2.78668	2.71714	1.99802	2.48721	2.27759
23045	198.566	65.2472	50.4789	46.4717	45.1363	44.5776

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

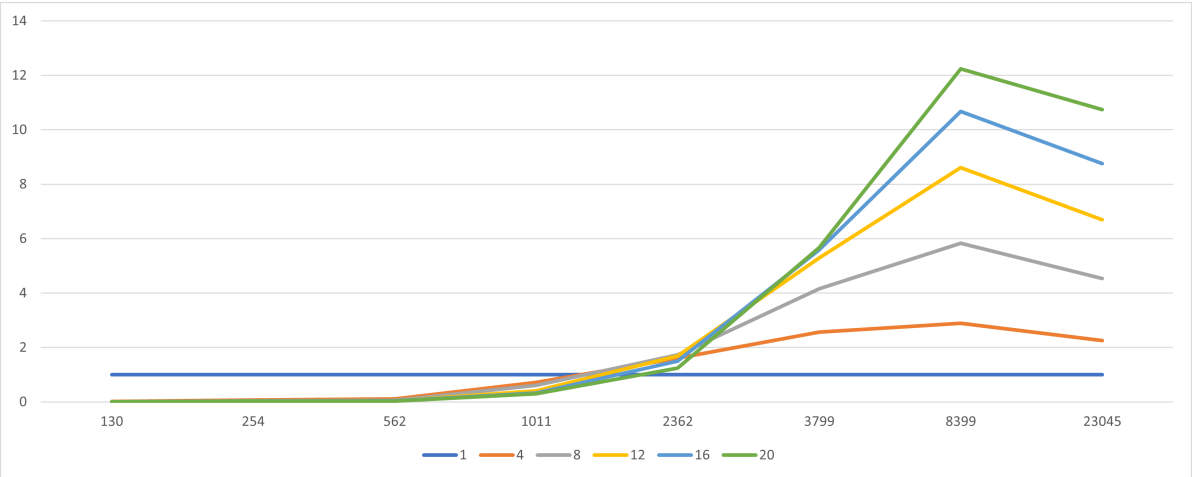

图 5.5: ARM 平台下特殊高斯消去算法加速比-输入数据规模（矩阵元素数量）

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

示：

矩阵列数 \ 线程数	1	4	8	12	16	20
130	6.20E-06	0.000657981	0.00151675	0.00180888	0.00214525	0.00263356
254	0.000203302	0.00264929	0.0041691	0.00517378	0.00733332	0.00892373
562	0.000545487	0.00422865	0.00666771	0.00898492	0.00953388	0.0146113
1011	0.00954177	0.0148444	0.0204453	0.0250353	0.028699	0.0353505
2362	0.0423403	0.0616395	0.0698515	0.070898	0.0820665	0.0889781
3799	0.692533	0.36458	0.346752	0.39223	0.487133	0.488261
8399	9.08002	2.78668	2.71714	1.99802	2.48721	2.27759
23045	198.566	65.2472	50.4789	46.4717	45.1363	44.5776

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

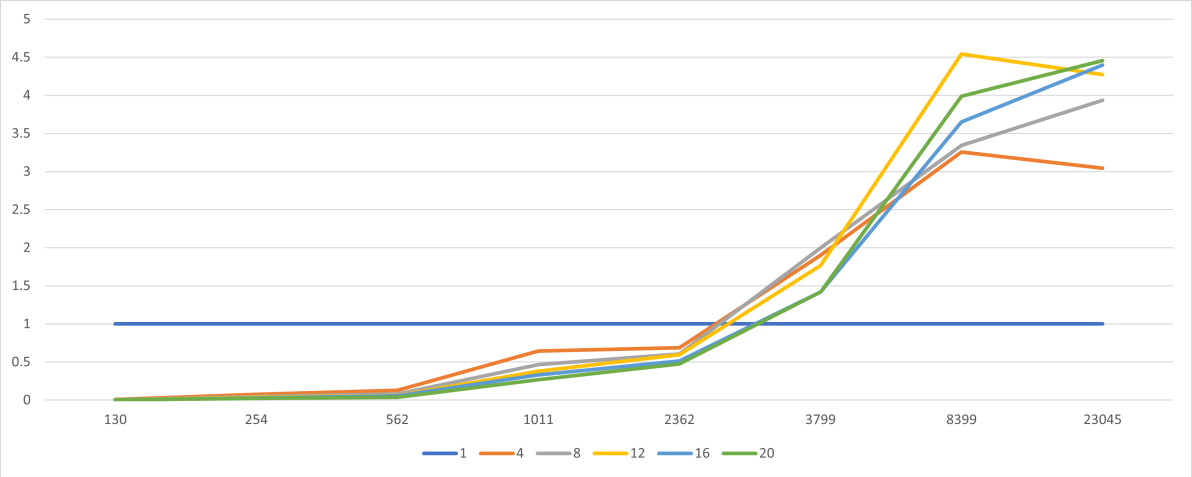

图 5.6: X86 平台下特殊高斯消去算法加速比-输入数据规模（矩阵列数）

6 结果分析

6.1 线程操作的性能消耗

如图6.7所示，每个循环步都进行线程创建和销毁工作的 Pthread 并行算法在运行时间上几乎与串行算法相近，而相比图5.3和图5.4所示的一次性创建线程的 Pthread 算法更是差别巨大。

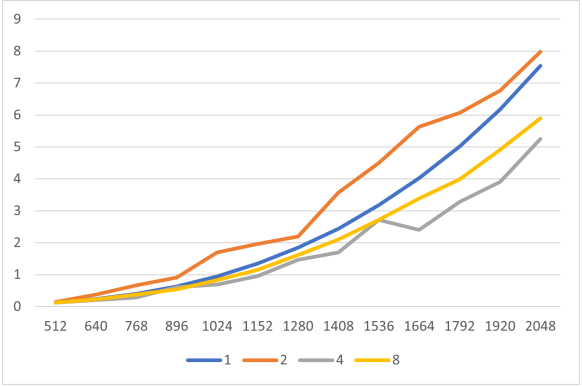

图 6.7: X86 平台下需要频繁创建线程的普通高斯消去算法运行时间-输入数据规模（矩阵列数）

通过 VTune 性能分析发现，在矩阵行数为 1024，线程数为 16 时，光是在 *pthread_create* 这一个函数就花费了 0.846s 的时间，而 SIMD 优化过的普通高斯消元算法总共也就运行了 0.940s。

以上现象说明频繁创建、销毁线程的开销是十分高昂的，我们尤其要避免在算法的循环步中创建线程。

此外，我们不难注意到在特殊高斯消去算法中，只有矩阵列数超过 2362 时，并行算法才开始有性能优势；而在普通高斯消去算法中，同样需要问题达到一定规模后才能让并行算法展现出优势。这证明了即便是一次性创建线程，同样也会有一定的性能消耗。同时，线程间通讯消耗等损耗也是不可忽略的。

6.2 DevCloud 上的性能限制

不难注意到，在图5.3和图5.4中，代表 Pthread 加速算法的曲线在图上比较集中，这意味着 X86 平台下，线程数和加速比明显不成正比；而在 ARM 平台下，如图5.3和图5.4所示，增加使用线程数基本能在加速比上得到正收益。

通过 Linux 下的性能检测工具 top 发现，当线程数超过 12 时，CPU 的占用率基本在 1000% 左右徘徊（top 中 CPU 占用率超过 100% 时表明程序正占用多个核心运行），而 DevCloud 平台使用的 CPU 是 24 核的 Xeon Gold 6128 处理器，因此作者怀疑 Devcloud 平台限制了单用户的性能。

在完成 OneAPI 投稿活动时，作者发现 OneAPI 改写的普通高斯消去算法在 DevCloud 上同样遇到了 1000% 的 CPU 占用瓶颈，因此这个瓶颈很大概率是 DevCloud 对用户做出一种限制。

6.3 负载不均

通过 Vtune 分析发现，线程数为 4，使用 7 号、矩阵列数为 8399 的测试数据集时，逻辑核心利用率为 2.804，物理核心利用率为 2.785；而相比之下，串行算法的逻辑核心利用率为 0.998，物理核心利用率 0.996。不难发现，串行算法的利用率已经十分接近 1，然而并行算法的核心利用率却与线程数 4 相去甚远。

对比串并行代码后，基本可以确认核心利用率的较低的主要原因是负载不均：由于作者采用的任务划分方法比较简单直接，因此，在运算过程中，特殊高斯消去算法经常会遇到已经升格的被消元行、或者当前列没有元素的情况，导致一整行都不需要消元。而部分线程可能分配到了大量一整行都不需要消元的被消元行，这样就导致了部分线程不能被完全利用的情况。

此外，特殊高斯消去算法中搜索可以充当消元子的被消元行这一过程也没有并行化，在主进程寻找消元子时，线程都被闲置了，这也是造成核心利用率低下的原因之一。