

南 开 大 学
NanKai University



pthread 编程实验——以高斯消去为例

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

2021年5月9日

摘要

本文使用所学 pthread 编程任务划分和同步机制，针对消去部分的两重循环，按照不同任务划分方式分别设计并实现 pthread 多线程算法。并考虑将其与 SIMD 算法结合，对各算法进行复杂度分析。同时借助 VTune profiling 等工具分析算法过程中的同步开销和空闲等待等。

关键字：高斯消去 pthread 同步 SIMD VTune

目录

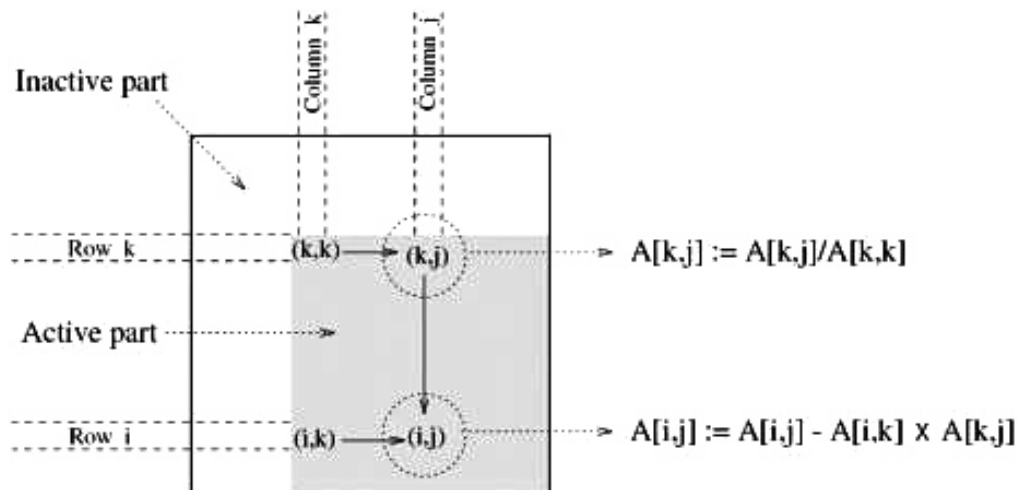
1	绪论	4
2	实验选题	5
3	算法设计与编程	6
3.1	初始化	6
3.2	pthread 编程	7
3.2.1	按行划分	8
3.2.2	按列划分	10
3.3	结合 SIMD	12
3.3.1	结合 SSE	13
3.3.2	结合 AVX	19
4	编译运行	22
5	结果分析	22
5.1	规模	23
5.2	算法复杂度分析	23
5.3	水平划分与垂直划分	23
5.4	SSE 与 AVX	25
5.5	线程数量对比	27
6	VTune 性能分析	28
7	总结	30

1 绪论

本文通过高斯消去法探究 pthread 多线程编程对于性能的提升。具体探究了使用按行划分与按列划分、结合 SSE 与 AVX、改变矩阵大小等不同情况下的时间性能以及加速比，还探究了算法优化的正确性。使用了性能剖析根据 VTune 对几种算法进行了具体的剖析，对比了不同算法下最终所执行的指令数，周期数，CPI 等等。

2 实验选题

高斯消去的计算模式如图所示，在第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 $k + 1$ 至 N 行进行减去第 k 行的操作。



高斯消去法（LU 分解）pthread 并行化为例，具体要求如下：

1. 使用 pthread 编程任务划分和同步机制，针对消去部分的两重循环，按照不同任务划分方式（如按行划分和按列划分），分别设计并实现 pthread 多线程算法。

2. 与 SIMD(SSE/AVX) 算法结合。

3. 改变矩阵的大小、线程数等参数，观测各算法运行时间的变化，对结果进行性能分析。借助 Vtune profiling 等工具分析算法过程中的同步开销和空闲等待等。

3 算法设计与编程

3.1 初始化

初始化系数矩阵：在使用高斯消去法的时候，有除法操作，为了避免出现除数为0以及无解的情况，将系数矩阵初始为上三角矩阵以便运算和验证。（N为系数矩阵规模大小）

```
1 // 初始系数矩阵，为了方便将其设置成上三角矩阵A
2 void init()
3 {
4     for (int i = 0; i < N; i++)
5     {
6         for (int j = i; j < N; j++)
7         {
8             A[i][j] = i + j + 1;
9         }
10    }
11 }
```

串行的高斯消去法代码如下：

```
1 // 高斯消去法（串行）
2 void Gauss_plain()
3 {
4     for (int k = 0; k < N; k++)
5     {
6         for (int j = k + 1; j < N; j++)
7         {
8             A[k][j] = A[k][j] / A[k][k];
9         }
10    A[k][k] = 1.0;
11    for (int i = k + 1; i < N; i++)
12    {
13        for (int j = k + 1; j < N; j++)
14        {
15            A[i][j] = A[i][j] - A[i][k] * A[k][j];
16        }
17    A[i][k] = 0;
```

```
18     }  
19 }  
20 }
```

3.2 pthread 编程

高斯消去过程中的计算集中在 4-5 的第一个内层循环（除法）和 8-13 行的第二个内层循环（双重循环，消去），对应矩阵右下角 $(n - k + 1) \times (n - k)$ 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，既可以采用水平划分（将其外层循环拆分，即每个线程分配若干行），也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。因此本次实验，只对 8-13 行的消去进行水平划分和垂直划分的线程分配，进行多线程算法测试。

通过信号量同步来避免频繁的创建和销毁线程。由主线程执行串行除法，工作线程执行消去。主线程开始时建立多工作线程；在每一轮消去过程中，工作线程先进入睡眠，主线程完成除法后将它们唤醒，自己进入睡眠；工作线程进行消去操作，完成后唤醒主线程，自己再进入睡眠；主线程被唤醒后进入下一轮消去过程，直至任务全部结束销毁工作线程。

```
1 bool flag; // 标志线程是否运行结束  
2  
3 typedef struct  
4 {  
5     int threadId;  
6     int startPos;  
7 } threadParm_t;
```

其中使用 flag 来标志是否还有任务执行；threadId 标志子线程的标号，startPos 标志最外层行数（即 k）。

```
1 // 信号量  
2 sem_t sem_parent;  
3 sem_t sem_children;  
4 sem_t sem_start;
```

每次处理到第 k 行时，需要设置好每个子线程的 `startPos` 之后再开始，通过信号量 `sem_start` 用来控制。主线程等待所有子线程都计算完毕后再进行下一次分配，子线程结束以后通过 `sem_parent` 通知主线程。`sem_children` 用于保证每一次工作结束以后达到初始状态，要等其他所有线程都结束，才可以开始新一轮，避免出现一些线程算的太快占用了两个 `sem_start` 的情况。

3.2.1 按行划分

按行划分 (`pt_row()`) 将行相减的操作交给不同的线程并行进行，比如一共 t 个线程运行时，第 i 个线程负责 $t * k + i$ 这些行的计算，每个子线程调用函数 `void *threadFunc_row(void *parm)`。

```
1 // 按行划分
2 void *threadFunc_row(void *parm)
3 {
4     threadParm_t *p = (threadParm_t *)parm;
5     while (true)
6     {
7         sem_wait(&sem_start);
8         // 如果已经没有任务执行，跳出循环结束
9         if (!flag)
10             break;
11         for (int i = p->startPos + p->threadId + 1; i < N; i +=
            NUMTHREADS)
12         {
13             for (int j = p->startPos + 1; j < N; j++)
14             {
15                 A[i][j] = A[i][j] - A[i][p->startPos] * A[p->startPos][j];
16             }
17             A[i][p->startPos] = 0.0;
18         }
19         // 每一次消去之后，唤醒主线程，工作线程睡眠
20         sem_post(&sem_parent);
```



```
21     sem_wait(&sem_children);
22 }
23 pthread_exit(NULL); // 返回结果给调用者
24 }
25
26 void pt_row()
27 {
28     threadParm_t threadParm[NUMTHREADS];
29     pthread_t thread[NUMTHREADS];
30     sem_init(&sem_parent, 0, 0);
31     sem_init(&sem_start, 0, 0);
32     sem_init(&sem_children, 0, 0);
33     // 开始时建立 NUMTHREADS 个工作线程
34     flag = true;
35     for (int i = 0; i < NUMTHREADS; i++)
36     {
37         threadParm[i].threadId = i;
38         pthread_create(&thread[i], NULL, threadFunc_row, (void *)&
39             threadParm[i]);
40     }
41     for (int k = 0; k < N; k++)
42     {
43         for (int j = k + 1; j < N; j++)
44         {
45             A[k][j] /= A[k][k];
46         }
47         A[k][k] = 1;
48         for (int i = 0; i < NUMTHREADS; i++)
49         {
50             threadParm[i].startPos = k;
51         }
52         for (int i = 0; i < NUMTHREADS; i++)
53         {
54             sem_post(&sem_start);
55         }
56         // 完成除法之后将工作线程唤醒，主线程睡眠
57         for (int i = 0; i < NUMTHREADS; i++)
58         {
```

```

58         sem_wait(&sem_parent);
59     }
60     for (int i = 0; i < NUMTHREADS; i++)
61     {
62         sem_post(&sem_children);
63     }
64 }
65 flag = false;
66 for (int i = 0; i < NUMTHREADS; i++)
67 {
68     sem_post(&sem_start);
69 }
70 for (int i = 0; i < NUMTHREADS; i++)
71 {
72     pthread_join(thread[i], NULL);
73 }
74 sem_destroy(&sem_start);
75 sem_destroy(&sem_children);
76 sem_destroy(&sem_parent);
77 }

```

3.2.2 按列划分

按行划分 (pt_col()) 将运算交给不同的线程并行进行，比如一共 t 个线程运行时，第 i 个线程负责 $t * k + i$ 这些列的计算，每个子线程调用函数 `void *threadFunc_col(void *parm)`。

```

1 // 按列划分
2 void *threadFunc_col(void *parm)
3 {
4     threadParm_t *p = (threadParm_t *)parm;
5     while (true)
6     {
7         sem_wait(&sem_start);
8         if (!flag)
9             break;
10        for (int i = p->startPos + 1; i < N; i++)
11        {

```

```

12         for (int j = p->startPos + p->threadId + 1; j < N; j +=
13             NUMTHREADS)
14         {
15             A[i][j] = A[i][j] - A[i][p->startPos] * A[p->startPos][j];
16         }
17     sem_post(&sem_parent);
18     sem_wait(&sem_children);
19 }
20 pthread_exit(NULL);
21 }
22
23 void pt_col()
24 {
25     threadParm_t threadParm[NUMTHREADS];
26     pthread_t thread[NUMTHREADS];
27     sem_init(&sem_parent, 0, 0);
28     sem_init(&sem_start, 0, 0);
29     sem_init(&sem_children, 0, 0);
30     // 开始时建立 NUMTHREADS 个工作线程
31     flag = true;
32     for (int i = 0; i < NUMTHREADS; i++)
33     {
34         threadParm[i].threadId = i;
35         pthread_create(&thread[i], NULL, threadFunc_col, (void *)&
36             threadParm[i]);
37     }
38     for (int k = 0; k < N; k++)
39     {
40         for (int j = k + 1; j < N; j++)
41         {
42             A[k][j] /= A[k][k];
43         }
44         A[k][k] = 1;
45         for (int i = 0; i < NUMTHREADS; i++)
46         {
47             threadParm[i].startPos = k;

```

```
47     }
48     for (int i = 0; i < NUMTHREADS; i++)
49     {
50         sem_post(&sem_start);
51     }
52     // 完成除法之后将工作线程唤醒，主线程睡眠
53     for (int i = 0; i < NUMTHREADS; i++)
54     {
55         sem_wait(&sem_parent);
56     }
57     // 主线程将第k列的元素全置为0，在子线程完成之后进行k+1列
58     for (int i = k + 1; i < N; i++)
59     {
60         A[i][k] = 0.0;
61     }
62     for (int i = 0; i < NUMTHREADS; i++)
63     {
64         sem_post(&sem_children);
65     }
66 }
67 flag = false;
68 for (int i = 0; i < NUMTHREADS; i++)
69 {
70     sem_post(&sem_start);
71 }
72 for (int i = 0; i < NUMTHREADS; i++)
73 {
74     pthread_join(thread[i], NULL);
75 }
76 sem_destroy(&sem_start);
77 sem_destroy(&sem_children);
78 sem_destroy(&sem_parent);
79 }
```

3.3 结合 SIMD

除法部分和消去的过程都可以使用 SSE/AVX 加速。需要注意的是，按

列划分的方法在使用 SSE/AVX 时，由于数据地址不连续，需要将矩阵元素一个个地装入寄存器中，而按行划分则利用了数据的连续性，可以直接将首地址作为参数传入。

3.3.1 结合 SSE

```

1 // 结合，按行划分SSE
2 void *threadFunc_sse_row(void *parm)
3 {
4     threadParm_t *p = (threadParm_t *)parm;
5     __m128 t1, t2, t3, t4;
6     while (true)
7     {
8         sem_wait(&sem_start);
9         if (!flag)
10            break;
11        int k = p->startPos;
12        for (int i = p->startPos + p->threadId + 1; i < N; i +=
            NUMTHREADS)
13        {
14            int remain = (N - k - 1) % 4;
15            // 不能凑够个的部分串行计算4
16            for (int j = k + 1; j <= k + remain; j++)
17            {
18                A[i][j] = A[i][j] - A[i][k] * A[k][j];
19            }
20            t1 = _mm_set1_ps(A[i][k]);
21            // 并行计算剩下的倍数个4float
22            for (int j = p->startPos + remain + 1; j < N; j += 4)
23            {
24                t2 = _mm_load_ps(A[k] + j);
25                t3 = _mm_mul_ps(t1, t2);
26                t4 = _mm_load_ps(A[i] + j);
27                t4 = _mm_sub_ps(t4, t3);
28                _mm_store_ps(A[i] + j, t4);
29            }
30            A[i][p->startPos] = 0.0;
31        }

```

```

32     // 每一次消去之后，唤醒主线程，工作线程睡眠
33     sem_post(&sem_parent);
34     sem_wait(&sem_children);
35 }
36 pthread_exit(NULL); //返回结果给调用者
37 }
38
39 void pt_sse_row()
40 {
41     threadParm_t threadParm[NUMTHREADS];
42     pthread_t thread[NUMTHREADS];
43     sem_init(&sem_parent, 0, 0);
44     sem_init(&sem_start, 0, 0);
45     sem_init(&sem_children, 0, 0);
46     // 开始时建立 NUMTHREADS 个工作线程
47     flag = true;
48     for (int i = 0; i < NUMTHREADS; i++)
49     {
50         threadParm[i].threadId = i;
51         pthread_create(&thread[i], NULL, threadFunc_sse_row, (void *)&
            threadParm[i]);
52     }
53     for (int k = 0; k < N; k++)
54     {
55         __m128 t1, t2;
56         // 不能凑够个的部分串行计算4
57         int remain = (N - k - 1) % 4;
58         for (int j = k + 1; j <= k + remain; j++)
59         {
60             A[k][j] = A[k][j] / A[k][k];
61         }
62         // 并行计算剩下的倍数个，减少使用除法4
63         t1 = _mm_set1_ps(1 / A[k][k]);
64         for (int j = k + remain + 1; j < N; j += 4)
65         {
66             t2 = _mm_load_ps(A[k] + j);
67             t2 = _mm_mul_ps(t1, t2);
68             _mm_store_ps(A[k] + j, t2);

```

```
69     }
70     A[k][k] = 1;
71     for (int i = 0; i < NUMTHREADS; i++)
72     {
73         threadParm[i].startPos = k;
74     }
75     for (int i = 0; i < NUMTHREADS; i++)
76     {
77         sem_post(&sem_start);
78     }
79     // 完成除法之后将工作线程唤醒，主线程睡眠
80     for (int i = 0; i < NUMTHREADS; i++)
81     {
82         sem_wait(&sem_parent);
83     }
84     for (int i = 0; i < NUMTHREADS; i++)
85     {
86         sem_post(&sem_children);
87     }
88 }
89 flag = false;
90 for (int i = 0; i < NUMTHREADS; i++)
91 {
92     sem_post(&sem_start);
93 }
94 for (int i = 0; i < NUMTHREADS; i++)
95 {
96     pthread_join(thread[i], NULL);
97 }
98 sem_destroy(&sem_start);
99 sem_destroy(&sem_children);
100 sem_destroy(&sem_parent);
101 }
102
103 // 结合，按列划分SSE
104 void *threadFunc_sse_col(void *parm)
105 {
106     threadParm_t *p = (threadParm_t *)parm;
```

```

107  __m128 t1, t2, t3, t4;
108  float f1[4], f2[4];
109  while (true)
110  {
111      sem_wait(&sem_start);
112      if (!flag)
113          break;
114      int k = p->startPos;
115      int remain = (N - k - 1) % 4;
116      for (int i = k + 1; i < k + 1 + remain; i++)
117      {
118          // 不能凑够个的部分串行计算4
119          for (int j = k + p->threadId + 1; j < N; j += NUM_THREADS)
120          {
121              A[i][j] = A[i][j] - A[i][k] * A[k][j];
122          }
123      }
124      for (int i = k + 1 + remain; i < N; i += 4)
125      {
126          // 并行计算剩下的倍数个4float
127          for (int j = k + p->threadId + 1; j < N; j += NUM_THREADS)
128          {
129              t1 = _mm_set1_ps(A[k][j]);
130              for (int m = 0; m < 4; m++)
131              {
132                  f1[m] = A[i + m][k];
133                  f2[m] = A[i + m][j];
134              }
135              t2 = _mm_load_ps(f1);
136              t3 = _mm_mul_ps(t1, t2);
137              t4 = _mm_load_ps(f2);
138              t4 = _mm_sub_ps(t4, t3);
139              _mm_store_ps(f1, t4);
140              for (int m = 0; m < 4; m++)
141              {
142                  A[i + m][j] = f1[m];
143              }
144          }

```



```

145     }
146     // 每一次消去之后，唤醒主线程，工作线程睡眠
147     sem_post(&sem_parent);
148     sem_wait(&sem_children);
149 }
150 pthread_exit(NULL); //返回结果给调用者
151 }
152
153 void pt_sse_col()
154 {
155     threadParm_t threadParm[NUM_THREADS];
156     pthread_t thread[NUM_THREADS];
157     sem_init(&sem_parent, 0, 0);
158     sem_init(&sem_start, 0, 0);
159     sem_init(&sem_children, 0, 0);
160     // 开始时建立 NUM_THREADS 个工作线程
161     flag = true;
162     for (int i = 0; i < NUM_THREADS; i++)
163     {
164         threadParm[i].threadId = i;
165         pthread_create(&thread[i], NULL, threadFunc_sse_col, (void *)&
            threadParm[i]);
166     }
167     for (int k = 0; k < N; k++)
168     {
169         __m128 t1, t2;
170         // 不能凑够个的部分串行计算4
171         int remain = (N - k - 1) % 4;
172         for (int j = k + 1; j <= k + remain; j++)
173         {
174             A[k][j] = A[k][j] / A[k][k];
175         }
176         // 并行计算剩下的倍数个，减少使用除法4
177         t1 = _mm_set1_ps(1 / A[k][k]);
178         for (int j = k + remain + 1; j < N; j += 4)
179         {
180             t2 = _mm_load_ps(A[k] + j);
181             t2 = _mm_mul_ps(t1, t2);

```

```
182         _mm_store_ps(A[k] + j, t2);
183     }
184     A[k][k] = 1;
185     for (int i = 0; i < NUMTHREADS; i++)
186     {
187         threadParm[i].startPos = k;
188     }
189     for (int i = 0; i < NUMTHREADS; i++)
190     {
191         sem_post(&sem_start);
192     }
193     // 完成除法之后将工作线程唤醒，主线程睡眠
194     for (int i = 0; i < NUMTHREADS; i++)
195     {
196         sem_wait(&sem_parent);
197     }
198     // 主线程将第列的行以下的元素全置为，在子线程完成之后进行kk0
199     for (int i = k + 1; i < N; i++)
200     {
201         A[i][k] = 0.0;
202     }
203     for (int i = 0; i < NUMTHREADS; i++)
204     {
205         sem_post(&sem_children);
206     }
207 }
208 flag = false;
209 for (int i = 0; i < NUMTHREADS; i++)
210 {
211     sem_post(&sem_start);
212 }
213 for (int i = 0; i < NUMTHREADS; i++)
214 {
215     pthread_join(thread[i], NULL);
216 }
217 sem_destroy(&sem_start);
218 sem_destroy(&sem_children);
219 sem_destroy(&sem_parent);
```

220 }

3.3.2 结合 AVX

由于按列访问结合 SIMD 的效率太低，AVX 就不测试按列访问的算法效率了。

```

1 // 结合，按行划分AVX
2 void *threadFunc_avx_row(void *parm)
3 {
4     threadParm_t *p = (threadParm_t *)parm;
5     __m256 t1, t2, t3, t4;
6     while (true)
7     {
8         sem_wait(&sem_start);
9         if (!flag)
10            break;
11        int k = p->startPos;
12        for (int i = p->startPos + p->threadId + 1; i < N; i +=
13            NUMTHREADS)
14        {
15            int remain = (N - k - 1) % 8;
16            // 不能凑够个的部分串行计算4
17            for (int j = k + 1; j <= k + remain; j++)
18            {
19                A[i][j] = A[i][j] - A[i][k] * A[k][j];
20            }
21            t1 = _mm256_set1_ps(A[i][k]);
22            // 并行计算剩下的倍数个4float
23            for (int j = p->startPos + remain + 1; j < N; j += 8)
24            {
25                t2 = _mm256_loadu_ps(A[k] + j);
26                t3 = _mm256_mul_ps(t1, t2);
27                t4 = _mm256_loadu_ps(A[i] + j);
28                t4 = _mm256_sub_ps(t4, t3);
29                _mm256_storeu_ps(A[i] + j, t4);
30            }
31            A[i][p->startPos] = 0.0;

```

```

31     }
32     // 每一次消去之后，唤醒主线程，工作线程睡眠
33     sem_post(&sem_parent);
34     sem_wait(&sem_children);
35 }
36 pthread_exit(NULL); //返回结果给调用者
37 }
38
39 void pt_avx_row()
40 {
41     threadParm_t threadParm[NUMTHREADS];
42     pthread_t thread[NUMTHREADS];
43     sem_init(&sem_parent, 0, 0);
44     sem_init(&sem_start, 0, 0);
45     sem_init(&sem_children, 0, 0);
46     // 开始时建立 NUMTHREADS 个工作线程
47     flag = true;
48     for (int i = 0; i < NUMTHREADS; i++)
49     {
50         threadParm[i].threadId = i;
51         pthread_create(&thread[i], NULL, threadFunc_avx_row, (void *)&
            threadParm[i]);
52     }
53     for (int k = 0; k < N; k++)
54     {
55         __m256 t1, t2;
56         // 不能凑够个的部分串行计算4
57         int remain = (N - k - 1) % 8;
58         for (int j = k + 1; j <= k + remain; j++)
59         {
60             A[k][j] = A[k][j] / A[k][k];
61         }
62         // 并行计算剩下的倍数个，减少使用除法4
63         t1 = _mm256_set1_ps(1 / A[k][k]);
64         for (int j = k + remain + 1; j < N; j += 8)
65         {
66             t2 = _mm256_loadu_ps(A[k] + j);
67             t2 = _mm256_mul_ps(t1, t2);

```

```
68         _mm256_storeu_ps(A[k] + j, t2);
69     }
70     A[k][k] = 1;
71     for (int i = 0; i < NUMTHREADS; i++)
72     {
73         threadParm[i].startPos = k;
74     }
75     for (int i = 0; i < NUMTHREADS; i++)
76     {
77         sem_post(&sem_start);
78     }
79     // 完成除法之后将工作线程唤醒，主线程睡眠
80     for (int i = 0; i < NUMTHREADS; i++)
81     {
82         sem_wait(&sem_parent);
83     }
84     for (int i = 0; i < NUMTHREADS; i++)
85     {
86         sem_post(&sem_children);
87     }
88 }
89 flag = false;
90 for (int i = 0; i < NUMTHREADS; i++)
91 {
92     sem_post(&sem_start);
93 }
94 for (int i = 0; i < NUMTHREADS; i++)
95 {
96     pthread_join(thread[i], NULL);
97 }
98 sem_destroy(&sem_start);
99 sem_destroy(&sem_children);
100 sem_destroy(&sem_parent);
101 }
```

4 编译运行

在 CodeBlocks 中编译运行：通过 QueryPerformance 来精确计时，同时运行 epoch=10 次取平均值。

操作系统环境： Windows 10 专业版 (64 位)

处理器： Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz (4 CPUs), 2.5GHz

RAM: 8.0GB

多线程的编译选项： -lpthread

SSE的编译选项： -march=corei7、-march=native

AVX的编译选项： -march=corei7-avx、-march=native

5 结果分析

先对上述算法的正确性进行验证，检验结果正确后，对不同规模、不同算法、不同线程下的性能进行测试并分析。

正确性判断代码如下：

```
1 // 正确性比较
2 bool comp(float A[N][N], float B[N][N])
3 {
4     for (int i = 0; i < N; i++)
5     {
6         for (int j = i; j < N; j++)
7         {
8             if (fabs(A[i][j] - B[i][j]) > 1e-6)
9             {
10                 cout << "误差超出10e, 误差为: -6";
11                 cout << fabs(A[i][j] - B[i][j]) << endl;
12                 return false;
13             }
14         }
15     }
16     cout << "误差未超出10e-6" << endl;
```

```
17     return true;  
18 }
```

5.1 规模

使用CPU-Z查看本机cache的信息如下：



因此分别设置测试规模 $N = 8, 16, 32, 64, 128, 256, 512, 1024, 2046$ 。

5.2 算法复杂度分析

矩阵规模为 n 时，高斯消去法串行算法的时间复杂度为 $O(n^3)$ ，在本次实验中，设 CPU 核心数为 n_1 ，线程数为 n_2 ，那么：内层循环的复杂度为 $O(n^2/\min(n_1, n_2))$ ，整体时间复杂度为 $O(n^2 + n^3/\min(n_1, n_2))$ ，再结合 SSE 方法的话，整体时间复杂度变为 $O((n^2 + n^3/\min(n_1, n_2))/4)$ ，使用 AVX 方法的话，整体时间复杂度变为为 $O((n^2 + n^3/\min(n_1, n_2))/8)$ 。

5.3 水平划分与垂直划分

针对不同规模下，水平划分与垂直划分的时间性能如下：

规模	平凡算法	按行划分	按列划分
8	0.00075	1.42362	1.17263
16	0.00524	1.83299	1.83824
32	0.0404	3.43514	2.98731
64	0.39745	5.9884	7.11629
128	3.0233	12.5519	11.7093
256	22.3963	26.3676	30.4229
512	176.826	98.5702	107.425
1024	1416.99	485.463	638.745
2048	11506.3	3907.27	4580.28

图 1: 水平划分与垂直划分时间比较

由此得到加速比如下:

规模	按行划分	按列划分
8	0.000526	0.00064
16	0.002859	0.002851
32	0.011761	0.0135
64	0.06637	0.05585
128	0.24086	0.2582
256	0.84939	0.736166
512	1.794	1.64604
1024	2.9188	2.2184
2048	2.9448	2.51214

图 2: 水平划分与垂直划分加速比比较

矩阵规模小于 512 时, 串行用时比所有并行策略的运行时间少得多。这是因为创建和销毁线程的代价很大, 数据量较小时, 这个代价远大于并行节省的时间, 所以从表中能看出对于较小的数据量 (如 8, 16), 串行的时间比并行少了 3 个数量级。矩阵规模大于 256 之后, 并行的优化效果逐渐显现, N 在 256 至 1024 之间时, 加速比随数据量的增加而增加, 这说明此时的数据量还不足, 一些子线程处于空闲状态。 N 在 1024 至 2048 之间时, 加速比基本变化不大, 这可能是因为每个子线程都能拿到充足的数据, 充分发挥多线程并行的优势。

按行划分的加速比几乎在所有情况下都高于按列划分的加速比，这是因为 C++ 按行储存矩阵，按列分配任务会导致访问的数据地址不连续，增加了 cache 缺失率，运行时间增长。

5.4 SSE 与 AVX

针对不同规模下，SSE 与 AVX 以及水平划分与垂直划分的时间性能如下：

规模	平凡算法	SSE+按行划分	SSE+按列划分	AVX+按行划分
8	0.00075	1.38434	1.54412	1.3684
16	0.00524	2.00587	2.33206	1.98679
32	0.0404	3.5543	3.44182	4.03703
64	0.39745	5.77013	6.85852	5.7382
128	3.0233	11.2552	13.757	11.0892
256	22.3963	23.8152	40.1145	21.876
512	176.826	71.9969	188.945	54.2348
1024	1416.99	301.326	1177.05	199.179
2048	11506.3	2263.73	8677.34	1631.93

图 3: 结合 SSE 与 AVX 下，水平划分与垂直划分时间比较

由此得到加速比如下：

规模	SSE+按行划分	SSE+按列划分	AVX+按行划分
8	0.0005412	0.0004857	0.0005481
16	0.002612	0.002247	0.0026374
32	0.0113665	0.011738	0.010007
64	0.06888	0.05795	0.069264
128	0.268614	0.21976	0.27263
256	0.9404	0.55831	1.02378
512	2.456	0.93586	3.26038
1024	4.7025	1.2038	7.1142
2048	5.0829	1.326	7.0507

图 4: 结合 SSE 与 AVX 下，水平划分与垂直划分加速比比较

当矩阵规模小于 512 时，SSE/AVX 的使用基本没有任何优化效果，这是因为数据量太小，并行的开销超过了并行的收益。当数据量增大时，加速效果明显，当数据大于 1024 时，SSE 能获得 5 倍左右的加速比，AVX 能获得 7 倍左右的加速比。

使用了 SSE/AVX 的同时按列划分，效果会更差：pt_sse_col 的加速比基本比 pt_col 还要低，这是因为 pt_sse_col 在计算时还需要将每一列的元素挨个取出放入 128 位寄存器，浪费了大量时间。

5.5 线程数量对比

测试线程数为2, 4, 8时, 按行访问与按列访问的加速比如下:

规模	2核		4核		8核	
	按行划分	按列划分	按行划分	按列划分	按行划分	按列划分
8	0.00075	0.000402	0.000526	0.00064	0.00352	0.000324
16	0.00377	0.00524	0.002859	0.002851	0.00167	0.001417
32	0.0157	0.0193	0.011761	0.0135	0.006447	0.00634
64	0.0579	0.10089	0.06637	0.05585	0.0377	0.0367
128	0.3448	0.32773	0.24086	0.2582	0.1559	0.156
256	0.8247	0.71722	0.84939	0.736166	0.5485	0.53883
512	1.3709	1.2527	1.794	1.64604	1.5143	1.2532
1024	1.65	1.5084	2.9188	2.2184	2.4868	1.9754
2048	1.8025	1.55308	2.9448	2.51214	2.868	1.595

图 5: 不同线程数下加速比

由此可以得到不同线程的效率:

规模	2核		4核		8核	
	按行划分	按列划分	按行划分	按列划分	按行划分	按列划分
8	0.0375%	0.0201%	0.01315%	0.016%	0.044%	0.00405%
16	0.1885%	0.262%	0.07148%	0.07128%	0.02088%	0.01771%
32	0.785%	0.965%	0.29403%	0.3375%	0.08059%	0.07925%
64	2.895%	5.0445%	1.65925%	1.3963%	0.4713%	0.45875%
128	17.24%	16.3865%	6.0215%	6.455%	1.9488%	1.95%
256	41.235%	35.861%	21.2348%	18.4042%	6.8563%	6.7354%
512	68.545%	62.635%	44.85%	41.151%	18.9288%	15.665%
1024	82.5%	75.42%	72.97%	55.46%	31.085%	24.6925%
2048	90.125%	77.654%	73.62%	62.804%	35.85%	19.9375%

图 6: 不同线程数的效率

可以看出, 当数据量充足 (大于 512) 时, 4 线程的加速比明显优于 2 线程, 8 线程的加速比与 4 线程的接近, 增加的子线程没能提高加速比。当子线程数量为 2 时, 效率较高, 且效率随数据规模的增大而提高。由于线程数越多, 额外开销就越多, 因此效率会下降。数据量充足时, 2 线程的效率能

达到 90%，4 线程的效率接近 70%，但是 8 线程的效率一直都很低，连 40% 都不到。因此 4 线程并行是比较合适的，线程较少不能充分地利用 CPU 核心，若创建过多的线程数，只能增加上下文切换的次数，因此会带来额外的开销。

6 VTune 性能分析

使用 Vtune 分析 `pt_row` 算法过程中的同步开销和空闲等待：对于 Microarchitecture Exploration，选择 Grouping: Function/thread/Logical Core/Call Stack，对于 4 个子线程、矩阵规模 $N=1024$ 的情况如下：可以看出运行

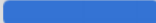








Function / Thread / Logical Core / Call Stack	CPU Time
▼ threadFunc_row<float [1024][1024]>	3.282s 
▶ Thread (TID: 9196)	0.819s 
▶ Thread (TID: 12304)	0.822s 
▶ Thread (TID: 9524)	0.822s 
▼ Thread (TID: 468)	0.819s 
▶ cpu_1	0.287s 
▶ cpu_2	0.259s 
▶ cpu_0	0.151s 
▶ cpu_3	0.122s 
▼ pt_row<float [1024][1024]>	0.005s
▶ Thread (TID: 10648)	0.005s

图 7: 线程运行时间

`threadFunc_row` 的四个线程的 TID 分别是 9196、12304、9524、468，运行时间分别为 0.819s、0.822s、0.822s、0.819s，负载比较均衡。

同时查看四个线程的开始时间和结束时间如下：

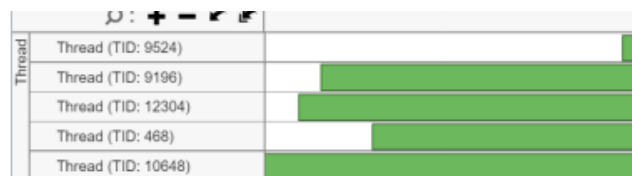


图 8: 四个线程开始时间

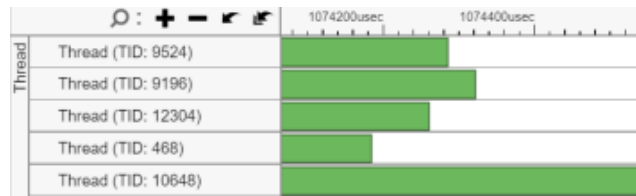


图 9: 四个线程结束时间

可以发现 TID 为 9524 的线程是最晚开始的，它应该是最晚拿到 start 信号量的线程，但是这个线程并不是最晚结束的，先于它结束的线程 468 与 12304 也没有很长时间的忙等待，因此这个算法还是比较好的。

测试单次运行 8 线程和 4 线程的运行时间和相关数据如下：

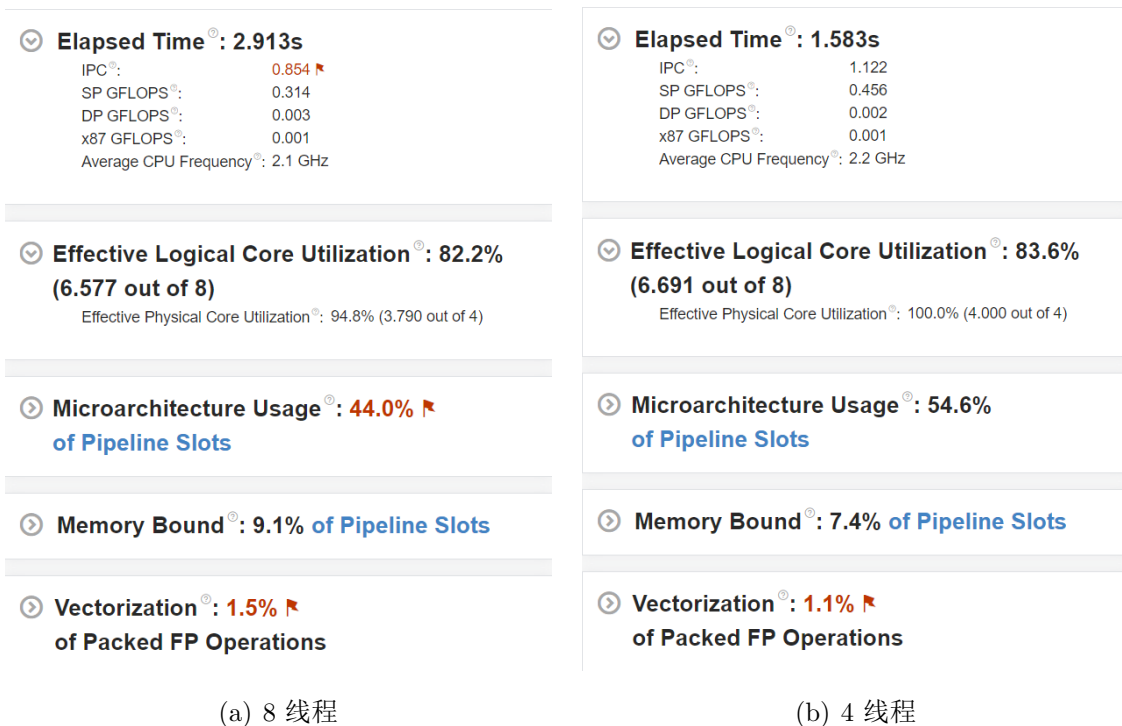


图 10: 8 线程与 4 线程的数据对比

可以发现，相对 8 线程，4 线程的有效的物理核心利用率更高，IPC 也更高，超过了1，因此增加的子线程没能提高资源利用率，也在一定程度上增加了线程开销，影响了性能。

查看 4 线程下，SSE 结合多线程的运行结果如下：

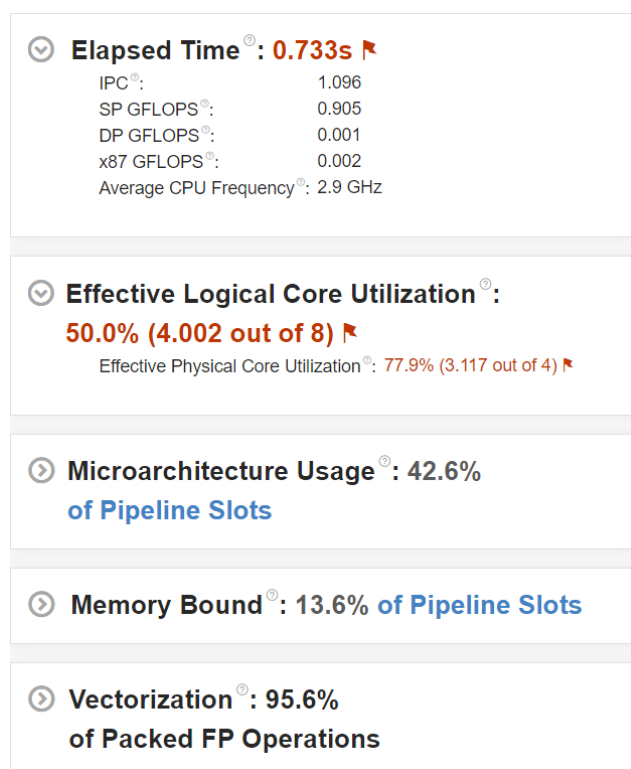


图 11: SSE 结合多线程的数据

可以发现，虽然运行时间大大减少了，但是 IPC 相对原来的 4 线程，反而下降了，这是因为使用 SSE 增加了指令数，而且有效的物理核心利用率也下降十分明显，不过可以发现指标 Vectorization 十分高，有 95.6%，向量化很高。因此规模较小的时候，使用多线程或者 SIMD 并行反而会增加一些线程消耗与指令消耗。

7 总结

这次实验理解了 pthread 实现多线程运行从而提高资源利用率。但是多线程并不是越多越好，线程为了提高资源使用效率来提高系统的效率，但是线程切换等等需要额外的开销，因此线程不是越多越好，在资源利用率达到一定程度时选择合适的线程数才是更好的。