南開大學
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计实验报告

高斯消去法 pthread 并行化实验报告

周辰霏 1712991

年级 : 2017 级

专业 : 计算机科学与技术

2020 年 4 月 17 日

摘要

针对高斯消元法 (LU 分解) 的 pthread 结合 SSE 和 AVX 的并行化实验。并对每一种算法进行基于 Vtune 的微观 Profiling 验证实验结果。

关键字：LU 分解；pthread 编程；SSE/AVX；Vtune

目录

一、 概述

(一) 问题描述

高斯消元法 (Gaussian Elimination) 是线性代数中的一个算法, 用于线性方程组求解、求矩阵的秩、求可逆方阵的逆矩阵。[1]

高斯消元法实际上就是最朴素的线性方程组消元, 其时间复杂度是 $O(n^3)$, 是一个在矩阵规模 N 较大时非常费时的算法, 且在高斯消元法的串行平凡算法中, 并非每一步都是必须的, 或者某几步是可以进行向量化的循环的。加之最近对于 pthread 编程的学习, 所以在此借助 pthread 以及 pthread 结合 SSE 和 AVX 的一系列优化达到 pthread 并行化的目的。

(二) 实验内容

针对高斯消去法 (LU 分解) 进行以下 pthread 并行化实现:

- 串行朴素 LU
- pthread 按行划分 LU 除法消去并行处理
- pthread 按行划分 LU 除法消去并行处理与 SSE/AVX 结合

并对各种并行化方法进行性能和参数上的比较包括但不限于数据测试规模、执行指令数、周期数、CPI、线程运行情况等的测算比对。

(三) 实验环境

- 系统环境: Windows10 家庭中文版 (64 位)
- 编译环境: gcc 8.1.0 x86_64-posit-seh, Codeblocks17.12, SSE3 指令, pthread
- 实验测试工具: Intel Vtune profiler 2020

二、 算法设计与实现

(一) 串行朴素 LU 实现

根据 Wiki 以及实验指导书, 串行朴素 LU 实际上就是将矩阵化为一个上三角矩阵, 其实现步骤为 (按行执行):

1. 第 k 步的时候从第 k+1 个元素开始除以第 k 个元素
2. 第 k 步的时候从第 k+1 个元素开始减去第 k 个元素与上一行第 k+1 个元素的乘积

串行朴素

```
1        void naive_lu(float mat[][N])
2        {
3            for (int k = 0; k < N; k++)
4            {
5                for (int j = k + 1; j < N; j++)// 除法
6                    mat[k][j] /=mat[k][k];
```

```
                mat[k][k] = 1.0;
                for (int i = k + 1; i < N; i++)// 减去第k行
                {
                    for (int j = k + 1; j < N; j++)
                        mat[i][j] -= mat[i][k] * mat[k][j];
                    mat[i][k] = 0;
                }
            }
        }
```

(二) pthread 按行划分除法消去并行处理

由于高斯消去法做了较多的重复同类运算, 故可以通过按行划分后对除法步骤和消去部分进行并行化多线程处理。此算法通过设置 barrier 同步对除法和消去部分进行同步多线程并行化。算法设计主要工作为: 在主函数中进行线程创建, 按行将前 k 行根据线程 ID 取线程总数的模划分给各个线程执行, 设置 barrier 等待除法执行完前 k 行之后再进行消去, 消去的划分方式同上, 同样等到全部消去完成后才通过 barrier 并退出线程。

pthread

```
void *row_pthread_lu(void *parm)    //pthread
{
        threadParm_t *p = (threadParm_t *)parm;// 存线程信息的数据结构
        int id = p->threadId;
    int start = id ;
        int end = id + 1;
        for (int k = 0; k < N; k++)
        {// 按取模结果划分给各线程
        if ((k + 1) % thread_num >= start
        && (k + 1) % thread_num < end)
        {
            for (int j = k + 1; j < N; j++)
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;
        }
        pthread_barrier_wait(&barrier1);
        //前k行除法完成就可以消去
        for (int i = k + 1; i < N; i++)
        {
            if (i % thread_num>= start && i % thread_num < end)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0;
            }
        }
                pthread_barrier_wait(&barrier2);
        }
```

```
        pthread_exit(NULL); // 返回退出
}
//main 函数中创建线程
for (long i = 0; i < thread_num; i++)
{
        threadParm[i].threadId = i;
        pthread_create(&threads[i],NULL,row_pthread_lu ,(void *)&threadParm[i]);
}
// 挂起等待新创建的线程完成
for (long i = 0; i < thread_num; i++)
        pthread_join(threads[i], NULL);
```

(三) pthread+SSE

在上述 pthread 按行划分算法的基础上对算法进行 SSE 优化, 即各类合并运算。在上一次实验减法并行化的基础上加入了除法的并行化。算法设计主要工作为: 中间变量引入及其数据类型更改 float→__m128、_mm_loadu_ps 加载数据到寄存器 _mm_storeu_ps 寄存器赋给变量、对循环进行 SSE 合并计算除法 _mm_div_ps、乘法 _mm_mul_ps 和减法 _mm_sub_ps。

pthread-SSE

```
void *row_pthread_sse_lu(void *parm)     //pthread+SSE
{
        threadParm_t *p = (threadParm_t *)parm;// 存线程信息的数据结构
        int id = p->threadId;
        int start = id ;
        int end = id + 1 ;
        __m128 t1, t2, t3;
        for (int k = 0; k < N; k++)
        {
                if ((k + 1) %thread_num>= start && (k + 1) % thread_num< end)
                {// 按取模结果划分给各线程
            float temp1[4] =
            { mat[k][k], mat[k][k], mat[k][k], mat[k][k] };
                        t1 = _mm_loadu_ps(temp1);
                        int j = k + 1;
                        for (j; j < N - 3; j += 4)// 除法
                        {
                                t2 = _mm_loadu_ps(mat[k] + j);
                                t3 = _mm_div_ps(t2, t1);
                                _mm_storeu_ps(mat[k] + j, t3);
                        }
                        for (j; j < N; j++)
                                mat[k][j] = mat[k][j] / mat[k][k];
                        mat[k][k] = 1.0;
                }
        pthread_barrier_wait(&barrier1);
        //前k行除法完成就可以消去
                for (int i = k + 1; i < N; i++)
```

```
                {
                        if (i % thread_num>= start && i % thread_num < end)
                        {// 消去
                float temp2[4] =
                { mat[i][k], mat[i][k], mat[i][k], mat[i][k] };
                                t1 = _mm_loadu_ps(temp2);
                                int j = k + 1;
                                for (j; j < N - 3; j += 4)
                                {
                                        t2 = _mm_loadu_ps(mat[i] + j);
                                        t3 = _mm_loadu_ps(mat[k] + j);
                                        t3 = _mm_mul_ps(t1, t3);
                                        t2 = _mm_sub_ps(t2, t3);
                                        _mm_storeu_ps(mat[i] + j, t2);
                                }
                                for (j; j < N; j++)
                    mat[i][j] =
                    mat[i][j] - mat[i][k] * mat[k][j];
                                mat[i][k] = 0;
                            }
                }
                pthread_barrier_wait(&barrier2);
        }
        pthread_exit(NULL);// 返回退出
}
//main 函数中创建线程
for (long i = 0; i < thread_num; i++)
{
        threadParm[i].threadId = i;
    pthread_create(&threads[i],NULL,row_pthread_sse_lu,(void *)&threadParm[i]);
}
// 挂起等待新创建的线程完成
for (long i = 0; i < thread_num; i++)
        pthread_join(threads[i], NULL);
```

（四） pthread+AVX

在上述 pthread 按行划分算法的基础上对算法进行 AVX 优化，思路与 SSE 一致，区别在于使用的指令不同。float→__m256，以及其他 mm 改为 mm256 即可，同时由于变为了 256 位，所以对齐需要 8 个原数据。

pthread+AVX

```
void *row_pthread_avx_lu(void *parm)// pthread+AVX
{
        threadParm_t *p = (threadParm_t *)parm;// 存线程信息的数据结构
        int id = p->threadId;
        int start = id ;
        int end = id + 1 ;
```

```
        __m256 t1, t2, t3;
        for (int k = 0; k < N; k++)
        {
                if ((k + 1) % thread_num >= start && (k + 1) % thread_num < end)
                {// 按取模结果划分给各线程
            float temp1[8] =
            { mat[k][k], mat[k][k], mat[k][k], mat[k][k],
            mat[k][k], mat[k][k], mat[k][k], mat[k][k] };
                        t1 = _mm256_loadu_ps(temp1);
                        int j = k + 1;
                        for (j; j < N - 7; j += 8)
                        {// 除法
                                t2 = _mm256_loadu_ps(mat[k] + j);
                                t3 = _mm256_div_ps(t2, t1);
                                _mm256_storeu_ps(mat[k] + j, t3);
                        }
                        for (j; j < N; j++)
                                mat[k][j] = mat[k][j] / mat[k][k];
                        mat[k][k] = 1.0;
                }
        pthread_barrier_wait(&barrier1);
        //前k行除法完成就可以消去
                for (int i = k + 1; i < N; i++)
                {
                        if (i % thread_num >= start && i % thread_num < end)
                        {
                float temp2[8] =
                { mat[i][k], mat[i][k], mat[i][k], mat[i][k],
                mat[i][k], mat[i][k], mat[i][k], mat[i][k] };
                                t1 = _mm256_loadu_ps(temp2);
                                int j = k + 1;
                                for (j; j < N - 7; j += 8)
                                {// 消去
                                        t2 = _mm256_loadu_ps(mat[i] + j);
                                        t3 = _mm256_loadu_ps(mat[k] + j);
                                        t3 = _mm256_mul_ps(t1, t3);
                                        t2 = _mm256_sub_ps(t2, t3);
                                        _mm256_storeu_ps(mat[i] + j, t2);
                                }
                                for (j; j < N; j++)
                    mat[i][j] =
                    mat[i][j] - mat[i][k] * mat[k][j];
                                mat[i][k] = 0;
                        }
                }
                pthread_barrier_wait(&barrier2);
        }
        pthread_exit(NULL);// 返回退出
```

```
55 }
56 //main 函数中创建线程
57 for (long i = 0; i < thread_num; i++)
58 {
59         threadParm[i].threadId = i;
60     pthread_create(&threads[i],NULL,row_pthread_avx_lu ,(void *)&threadParm[i]);
61 }
62 //挂起等待新创建的线程完成
63 for (long i = 0; i < thread_num; i++)
64         pthread_join(threads[i], NULL);
```

三、 高精度计时分析算法复杂性

由于实验中可能出现种种误差, 为尽量保证实验的公平性, 作以下假设:

- Cache 对所有算法都是公平的, 均为按行获取按行划分, 每次换算法之前通过 reset 函数令初始矩阵一致, 并验证结果正确性, 在最终结果一致 (以串行朴素 LU 的计算结果为基准) 的情况下才进行高精度计时
- 由于 N 较小的时候测出的计算时间也较小, 误差较大, 故采取多次测量取均值的方法确定较合理的性能测试结果, 同时保证几种算法重复次数一致, 减少误差
- 为保证一致性, 使用 barrier 同步机制, 在前 k 个执行结束之后才放行就可以保证结果的正确性。

同上一次实验, 本次的测试区间比较疏松, 但是仍考虑 cache 大小的情况进行 N 的取值设定。根据下图可以得知本机为 4 核 8 线程, 故线程池总数取取 4、8。同步开销测算的是两次 barrier 的到达时间与结束时间的差值, 即多线程的等待时间测算。

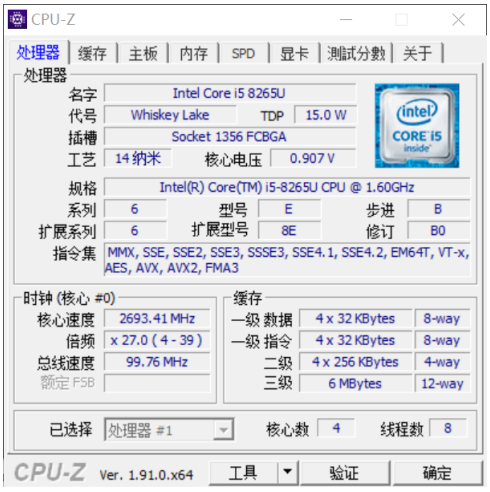

图 1: CPU 核心数

根据全部测试数据绘制运行时间、同步开销、加速比对比图、效率对比图, 运行时间图可以看出各种方法对于朴素方法都节约了近一半的运行时间, 四线程和八线程运行时间没有明显差异, 整体上辅以 SSE 的加速效果最好。原因可能是创建更多的线程耗费的时间以及同步开销更大, 所以运行时间没有明显提升, 而 AVX 打包和解包的代价相较于 SSE 大得多。

		naive	pthread(4)	p-SSE(4)	p-AVX(4)	pthread(8)	p-SSE(8)	p-AVX(8)
16	运行时间	0.00093	0.52202	0.50492	0.77808	0.93495	0.90386	0.7828
	同步开销		0.34825	0.11879	0.19575	0.320099	0.302729	0.24838
64	运行时间	0.0604	1.36181	1.29533	1.49047	2.65725	2.14262	2.0546
	同步开销		1.0039	0.427267	0.48152	2.12733	0.891829	0.76537
100	运行时间	0.21696	1.60313	1.38298	1.45144	2.96663	2.875	3.17416
	同步开销		1.41146	0.617823	0.56498	2.65636	1.65636	1.57147
200	运行时间	1.98219	3.65478	3.83555	4.05119	6.88497	5.9308	6.21729
	同步开销		3.32381	1.34865	2.022	6.47566	2.74084	2.87431
300	运行时间	6.06507	7.61351	6.19877	6.24898	13.3045	10.5858	9.86365
	同步开销		6.91254	2.56901	2.4516	10.4685	4.5699	4.21534
400	运行时间	13.6391	11.6537	8.70426	8.99385	17.9963	13.1117	15.8774
	同步开销		9.19258	2.2656	3.13501	16.153	5.60489	6.41141
500	运行时间	30.0577	17.8751	11.7712	13.0496	22.8858	17.8	16.9437
	同步开销		11.9239	4.07166	5.02669	19.9403	8.03474	6.77541
800	运行时间	115.032	46.5607	24.9022	26.664	53.5272	34.3954	34.6666
	同步开销		23.9262	6.99874	6.58797	47.5177	13.4971	13.8265
1000	运行时间	223.376	84.5977	42.069	42.2429	95.3655	44.8235	46.7985
	同步开销		33.7215	9.32275	10.3357	88.738	15.8686	18.0072
2000	运行时间	1978.87	820.334	735.305	775.012	786.265	715.017	755.41
	同步开销		133.938	51.4321	55.7575	185.897	33.2472	35.7298

表 1: 性能测试结果 (单位:ms)

加速比对比图则可以看出各个算法在加速上的趋势基本一致, 最高可达 5.5, 但是是比较反常的现象。N>400 以后加速比均大于 1, 除去反常现象整体上维持在 [2,3] 之间。八线程在 N=1500 以后优于四线程,SSE 和 AVX 的趋势与上一次实验一致,N<2000 都是 SSE 更优, 之后则是 AVX 更优。

效率对比图, 整体上四线程效率优于八线程, 课件四核八线程并不是真的可以很好的完成八线程的任务 (同步开销, 线程开销更大), 同运行时间反映的一致,SSE 效率更高,AVX 次之, 单独使用 pthread 最次。

同步开销占总运行时间比例图中, 同样是八线程占比高于四线程,N 较大时 AV 线占比高于 SSE, 此图反映的是线程同步所耗费的时间, 其实从最高占比达 0.9 就可以看出来, 线程的执行完成时间是不均匀的, 所以导致了同步开销时间在总运行时间中占比较高。也可以看出 N 越大占比越小, 可能是由于 N 越大计算量越大, 各线程分配也更均匀, 总的同步开销就越来越小。

整体来看,pthread+SSE+ 四线程的效果最好, 原因是四核 CPU 对八线程的处理并不能像八核那样理想,SSE 在打包解包上的开销小于 AVX, 故这一组合相对效果更好。

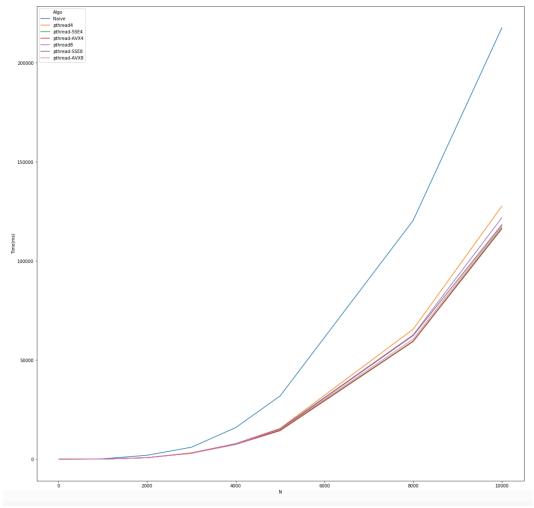

图 2: 运行时间

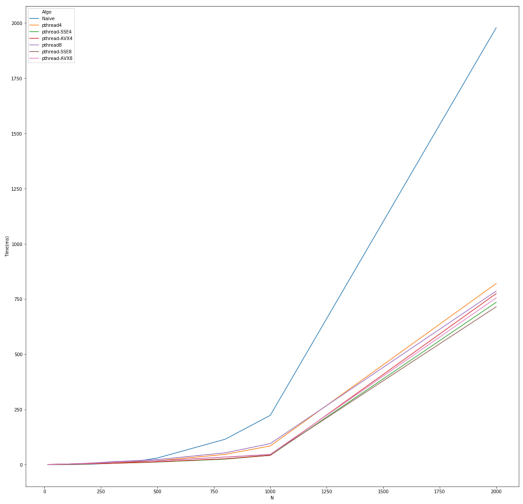

图 3: 运行时间 (1-2000)

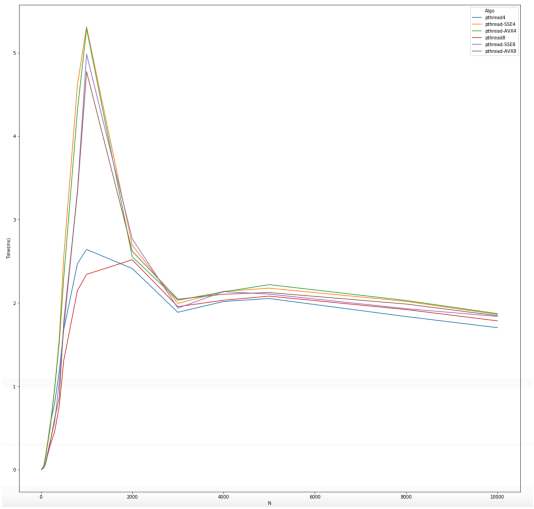

图 4: 加速比

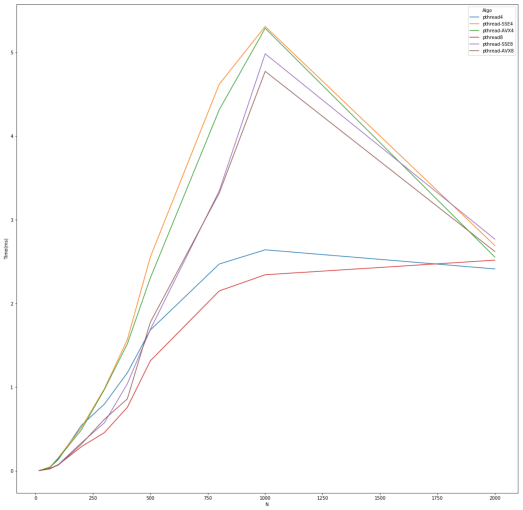

图 5: 加速比 (1-2000)

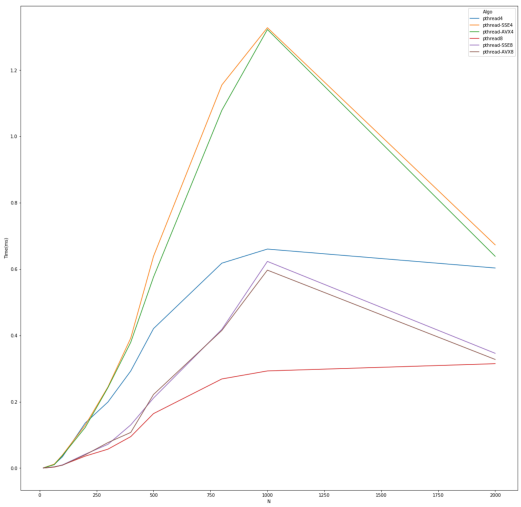

图 6: 效率

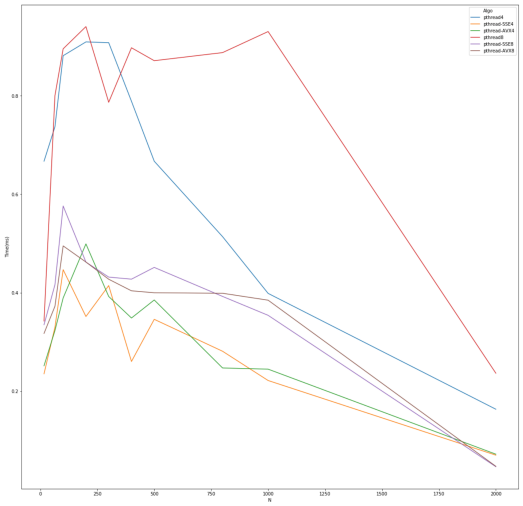

图 7: 同步开销占运行时间比

(一) 基于 Vtune 的程序性能剖析

由于这次的编程侧重点是编译后指令上的优化, 所以使用 Vtune profiling 的侧重点在微体系结构的指令数、周期数以及 CPI 以及线程执行情况上, 故按上次实验进行相同配置即可, 由于这次测试用例规模的变动不是重点, 所以 CPU 间隔取 0.1ms 即可。

如图为 N=2000 时四线程的 Profiling, 其他 N 取值表现相似。

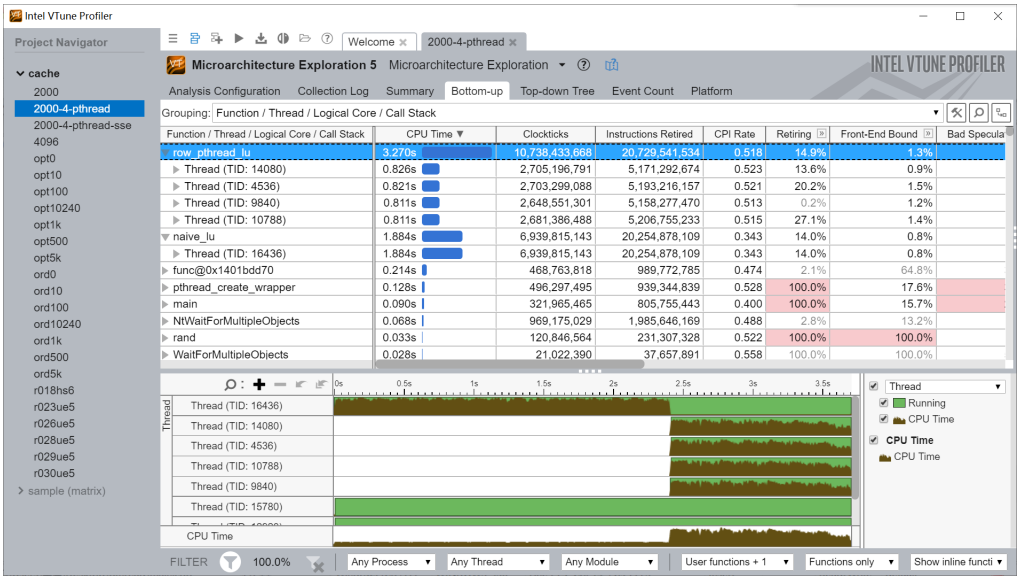

图 8: N=2000-4pthread

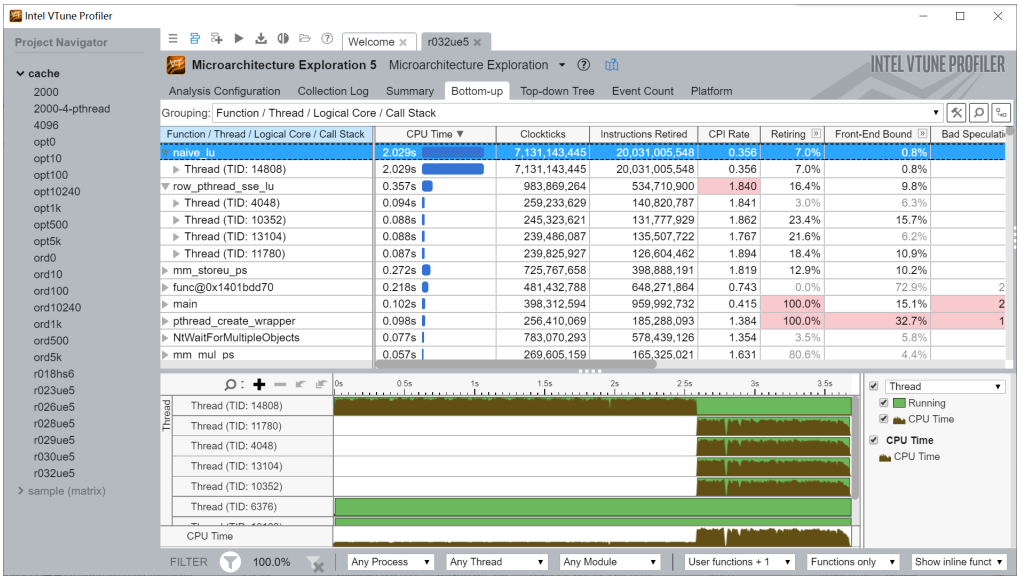

图 9: N=2000-4pthread-SSE

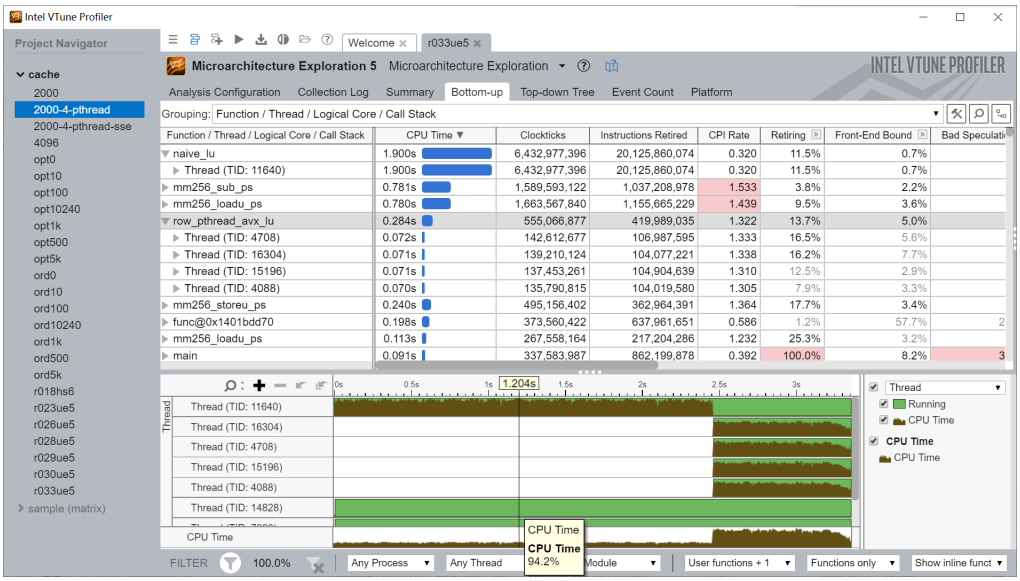

图 10: N=2000-4pthread-AVX

由于线程以及 barrier 的销毁指令在代码最后, 所以线程的绿色部分运行时间并看不出来太多重点, 但是结合 CPU time 可以看出正如我们所预料的一样, 到了 N 较大的时候, 各个线程的计算时间趋于一致。与实测不一样的是, 这里的 AVX 在周期数、指令数和 CPI 上的表现要比 SSE 好。在对 N 较小 (N≤100) 进行测试后, 发现此时对于各种 pthread 的优化每个函数都只拥有一个线程且这个线程大多数时间都处于等待状态, 且创建线程耗费时间多所以导致了这些 N 的取值下 pthread 的优化不仅运行时间变长了,CPI 也更差了。而对于 N=1000 的反常情况, 在 vtune 下更为明显, 加速比接近达到 7, 可能是因为串行指令数是并行的 20 几倍 (非主要原因), 而 cache 命中率并行算法恰好高于串行算法的结果, 使得通过节省访存开销提升了性能。

而对于四核八线程导致的八线程跟四线程的优化效果一致问题, 根据 Vtune 的结果, 可以发现其实在四线程的时候对每一个线程的操作也是通过八个虚拟核实现的, 所以四线程和八线程的效果基本上是一致的。

四、 总结

这个实验要借助对信号量同步互斥等内容的理解再结合上次实验的 SSE 以及 AVX 方法才可以实现, 但是最终的结果仍然和预想不完全一致。实验算是让我对于多线程的编程有了初步以及本质的了解, 了解到 pthread 并行化的优势以及不足。

参考文献

[1] 高斯消元法 [EB/OL].https://zh.wikipedia.org/wiki/高斯消去法

[2] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE3,AVX&expand=3931

[3] 课件 algo1-2,pthread3-6