南開大學
Nankai University

计算机学院

并行程序设计实验报告报告

高斯消去的 OpenMP 优化

姓名： 高祎珂

学号：2011743

专业：计算机科学与技术

2022 年 5 月 21 日

目录

1 实验介绍

1.1 实验简介

对于普通高斯消去在之前的实验中有过详细介绍，此处不再叙述，OpenMP 其实我觉得就是自动的 Pthread，我们不需要自己给线程分配任务，只需要指定要并行的区域，然后调用 openMP 函数就可以实现多线程。普通高斯消去的二重循环和三重循环内部计算没有顺序影响，可以并行计算，所以可以进行多线程优化，而多线程与 SIMD 比较起来，我认为，其优势在于可以不同的线程执行不同的函数操作，但是再次实验中并没有体现出来，不过对于多线程我们可以自己指定线程数，而不用局限于 SSE 等语言所限定的固定向量化操作数。

1.2 实验设计

1. 首先生成后续进行高斯消去的矩阵，编写代码实现高斯消去串行算法。
2. 在 ARM 平台上，进行 openmp 实验，进行静态任务划分和动态任务划分运行时间的对比，并将 SIMD 加入程序，观察加入 SIMD 后的效果，基于 SIMD 实验，三重循环向量化的效果比较好，所以此次仅进行三重循环的向量化，。
3. 进行 perf 分析，查看不同线程的运行时间。
4. 在 X86 平台上进行同样操作，分析在 X86 平台上的优化表现，并且利用 VTune 工具进行更细致的分析，最后比较一下两平台。

1.3 理论分析

如果开辟出 n 个子线程，不考虑开辟销毁线程过程中的损失以及其他消耗，理论上应该能达到 n 倍的加速比，但是由于线程的开销，达不到这么高的优化，而 openMP 线程开销有多大，这需要通过实验来验证。动态任务划分线程空闲时间更短，理论上时间会更短同伙实验来验证是否会如此。另外加上 SIMD 优化，根据上次实验，可以大致推算能达到 2 倍左右的效率提升，接下来根据实验来验证假设。

2 ARM 平台

2.1 算法设计

完整代码上传到了github，这里进行核心代码的展示。

omp 静态划分

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    for (int k = 0; k < n; ++k) {
        // 串行部分
#pragma omp single
        {
            float tmp = A[k][k];
            for (int j = k + 1; j < n; ++j)
```

```
            {
                A[k][j] = A[k][j] / tmp;
            }
            A[k][k] = 1.0;
        }
        // 并行部分，使用行划分
#pragma omp for schedule(static,NUM_THREADS)
        for (int i = k + 1; i < n; ++i)
        {
            float tmp = A[i][k];
            for (int j = k + 1; j < n; ++j)
            {
                A[i][j] = A[i][j] - tmp * A[k][j];
            }
            A[i][k] = 0.0;
        }
        // 离开for循环时，各个线程默认同步，进入下一行的处理
    }
}
```

omp 动态划分

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    for (int k = 0; k < n; ++k) {
        // 串行部分
#pragma omp single
        {
            float tmp = D[k][k];
            for (int j = k + 1; j < n; ++j)
            {
                D[k][j] =D[k][j] / tmp;
            }
            D[k][k] = 1.0;
        }
        // 并行部分，使用行划分
#pragma omp for schedule(dynamic,NUM_THREADS)
        for (int i = k + 1; i < n; ++i)
        {
            float tmp = D[i][k];
            for (int j = k + 1; j < n; ++j)
            {
                D[i][j] = D[i][j] - tmp *D[k][j];
            }
            D[i][k] = 0.0;
        }
    }
}
```

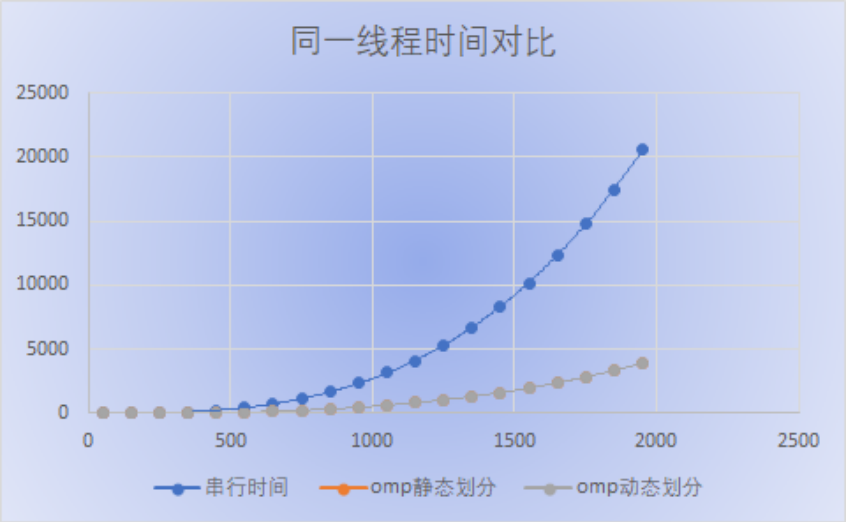
omp+SIMD

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    for (int k = 0; k < n; ++k) {
        // 串行部分
#pragma omp single
        {
            float tmp = B[k][k];
            for (int j = k + 1; j < n; ++j)
            {
                B[k][j] = B[k][j] / tmp;
            }
            B[k][k] = 1.0;
        }
        // 并行部分，使用行划分
#pragma omp for schedule(static,NUM_THREADS)
        for (int i = k + 1; i < n; ++i)
        {
            float32x4_t vaik = vdupq_n_f32(B[i][k]);
            int j = k + 1;
            for (j = k + 1; j + 4 <= n; j += 4)
            {
                float32x4_t vakj = vld1q_f32(&B[k][j]);
                float32x4_t vaij = vld1q_f32(&B[i][j]);
                float32x4_t vx = vmulq_f32(vakj, vaik);
                vaij = vsubq_f32(vaij, vx);

                vst1q_f32(&B[i][j], vaij);

            }
            while (j < n)
            {
                B[i][j] -= B[k][j] * B[i][k];
                j++;
            }
            B[i][k] = 0;
        }
        // 离开for循环时，各个线程默认同步，进入下一行的处理
    }
}
```

2.2 不同规模下结果分析

这是在设置线程为 6 的条件下进行的不同规模结果分析

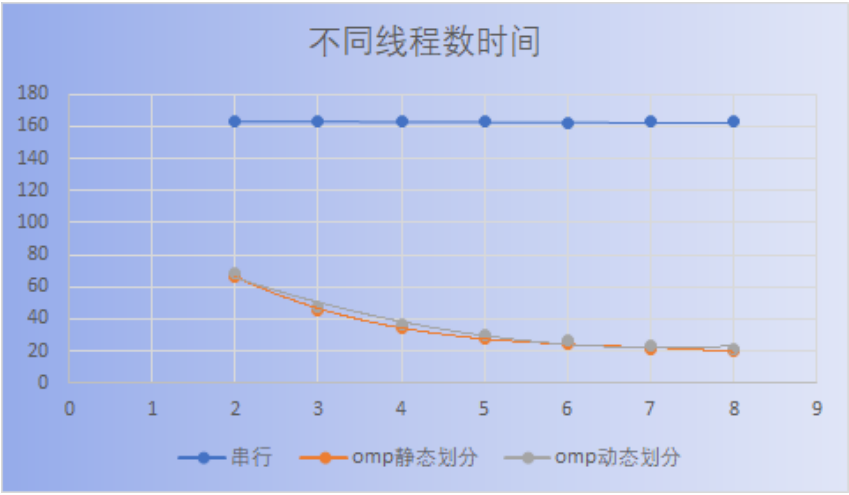
数据规模	串行/ms	静态划分/ms	动态划分/ms
50	0.32595	0.22073	0.16824
150	8.64828	2.05935	2.25708
250	39.92001	8.74644	9.31234
350	109.4569	23.47666	24.7407
450	236.9933	48.93479	51.32491
550	439.2537	88.43326	92.5396
650	738.7353	145.316	152.0244
750	1139.664	225.353	234.2742
850	1657.084	326.8184	336.5728
950	2335.691	456.1426	468.1699
1050	3163.977	615.0381	627.0876
1150	4090.64	822.3891	822.7627
1250	5316.301	1033.172	1039.704
1350	6730.009	1306.503	1307.781
1450	8340.069	1609.259	1615.544
1550	10184.23	1979.399	1978.885
1650	12366.94	2358.048	2372.578
1750	14742.11	2820.01	2830.846
1850	17451.33	3323.188	3333.264
1950	20543.7	3922.358	3928.678



此结果是设置子线程数为 6 时得到的，优化效果较为显著，上次做实验就发现了，鲲鹏的线程开销好像较小，openmp 的加速比比 X86 效果好，这次开 6 个线程，加速比达到了近 5 倍，还是比较理想的。下面看一下在鲲鹏上不同线程数时时间的对比。

2.3 不同线程数下结果分析

子线程数	串行/ms	静态划分/ms	动态划分/ms
8	162.6698	19.82838	21.58082
7	162.7289	21.5609	23.43835
6	162.5966	24.2089	26.53063
5	162.7224	28.16076	29.97821
4	162.7351	34.8044	36.65372
3	162.7065	45.69878	47.59495
2	162.6799	66.87046	68.14027



基于上述实验，这里选择了矩阵规模为 400 进行实验测试，基于鲲鹏一个计算核心有 8 个线程，从实验数据可以看出，随着线程数的减少，omp 函数运行时间都在不断增加，omp 基本实现了开辟 n 个线程，实现 n 倍加速比，但是鲲鹏我不知道是什么原因，有的线程优化甚至超过了开辟的线程数，而且在我设置线程数为 1 的时候，omp 程序也会有一定的优化，这个我还没搞明白。

2.4 Pthread+SIMD

这是加入 simd 的函数，完整代码上传到了github

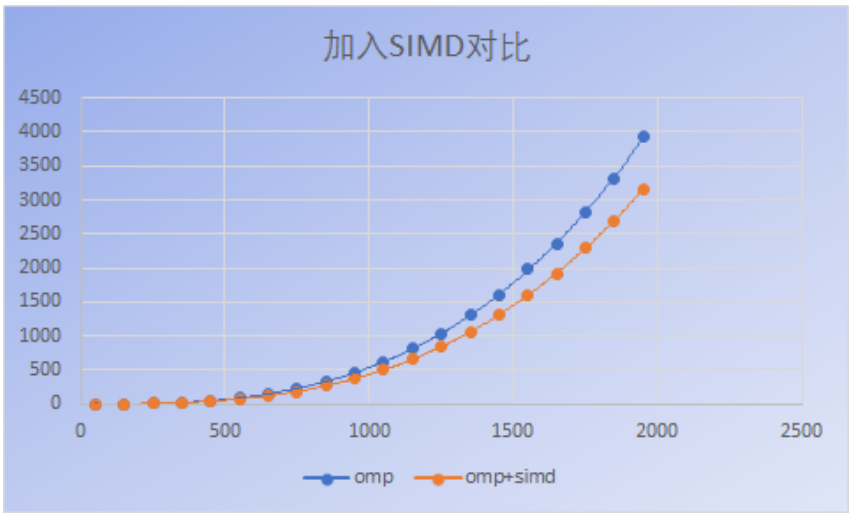
三重循环 SIMD

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    for (int k = 0; k < n; ++k) {
        // 串行部分
#pragma omp single
        {
            float tmp = B[k][k];
            for (int j = k + 1; j < n; ++j)
            {
                B[k][j] = B[k][j] / tmp;
            }
            B[k][k] = 1.0;
        }
```

```
            // 并行部分，使用行划分
    #pragma omp for schedule(static,NUM_THREADS)
            for (int i = k + 1; i < n; ++i)
            {
                float32x4_t vaik = vdupq_n_f32(B[i][k]);
                int j = k + 1;
                for (j = k + 1; j + 4 <= n; j += 4)
                {
                    float32x4_t vakj = vld1q_f32(&B[k][j]);
                    float32x4_t vaij = vld1q_f32(&B[i][j]);
                    float32x4_t vx = vmulq_f32(vakj, vaik);
                    vaij = vsubq_f32(vaij, vx);
                    vst1q_f32(&B[i][j], vaij);
                }
                while (j < n)
                {
                    B[i][j] -= B[k][j] * B[i][k];
                    j++;
                }
                B[i][k] = 0;
            }
            // 离开for循环时，各个线程默认同步，进入下一行的处理
        }
    }
```

运行得到的结果如下：

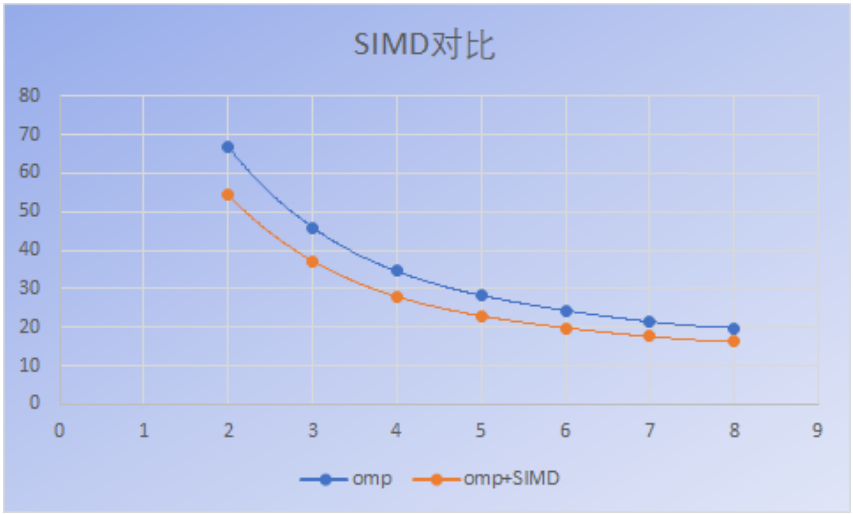
数据规模	静态划分/ms	静态划分 +simd/ms
50	0.22073	0.11904
150	2.05935	1.69316
250	8.74644	7.21673
350	23.47666	19.0415
450	48.93479	39.64693
550	88.43326	71.69252
650	145.316	117.8707
750	225.353	181.7353
850	326.8184	265.0795
950	456.1426	369.6528
1050	615.0381	500.6139
1150	822.3891	658.6824
1250	1033.172	841.2299
1350	1306.503	1057.584
1450	1609.259	1306.042
1550	1979.399	1598.938
1650	2358.048	1922.086
1750	2820.01	2292.973
1850	3323.188	2698.513
1950	3922.358	3161.102



这里线程数依旧设置为 6，加入 SIMD 整体变化规律与只进行 omp 优化相同，SIMD 是进行了三重循环的 SIMD（基于上次实验，三重循环向量化的效果更显著），基于上次实验，可以得出能达到 2 倍左右的优化效果，但是对于此实验而言，似乎并没有如此大的优化效果，只有 1.5 倍左右的优化，我觉得可能是因为线程占用资源较大，导致本应该的向量化但其实只是进行了虚拟的向量化。

下面看一下不同线程数，矩阵规模在 400 时的加入 SIMD 的优化，下面是结果

子线程数	静态划分/ms	静态划分 +simd/ms
8	19.82838	16.2045
7	21.5609	17.64112
6	24.2089	19.76883
5	28.16076	22.87702
4	34.8044	28.08859
3	45.69878	37.13081
2	66.87046	54.33839



可以从图中看到，当线程数较少时 SIMD 的优化效果较为明显，但是 SIMD 整体的优化效果并没有之前单独进行 SIMD 的效果好。

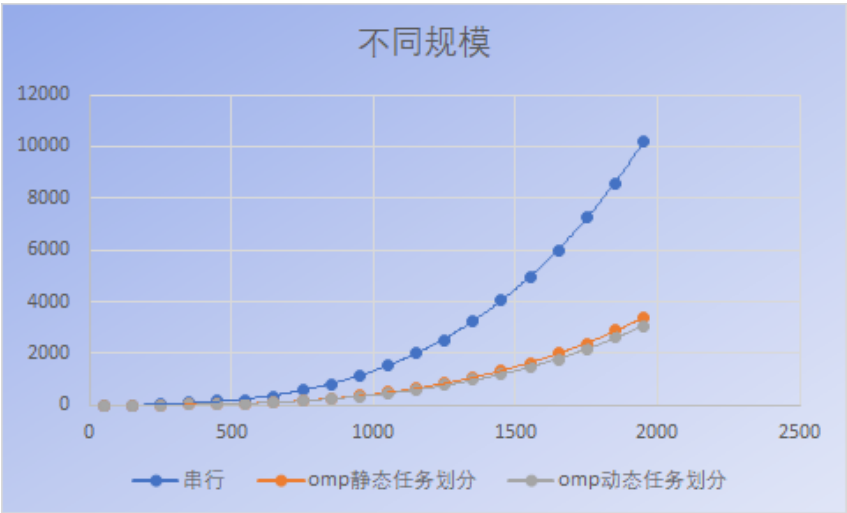
3 X86 平台

3.1 算法设计

与 ARM 平台的算法设计基本相同，只是在 SIMD 设计时更换一下语言，本次实验只用了 SSE 指令集，源码链接为github, 这里不再进行代码的展示。

3.2 不同规模下结果分析

数据规模	串行/ms	静态划分/ms	动态划分/ms
50	0.1942	2.5984	0.4871
150	6.8245	3.0979	4.2405
250	32.1758	8.6661	8.5054
350	103.151	30.4849	23.2818
450	208.4	64.3559	61.3921
550	216.566	70.7068	68.4943
650	358.56	112.853	108.924
750	575.732	174.596	161.635
850	835.527	259.009	245.941
950	1159.9	363.808	333.378
1050	1575.74	506.568	460.508
1150	2024.06	645.787	597.298
1250	2537.79	845.266	796.993
1350	3225.67	1094.85	986.787
1450	4087.07	1351.8	1232.98
1550	4985.23	1651.57	1463.35
1650	5999.97	2007.88	1761.22
1750	7258.98	2382.1	2140.04
1850	8571.67	2888.29	2638.62
1950	10191.3	3387.64	3078.22

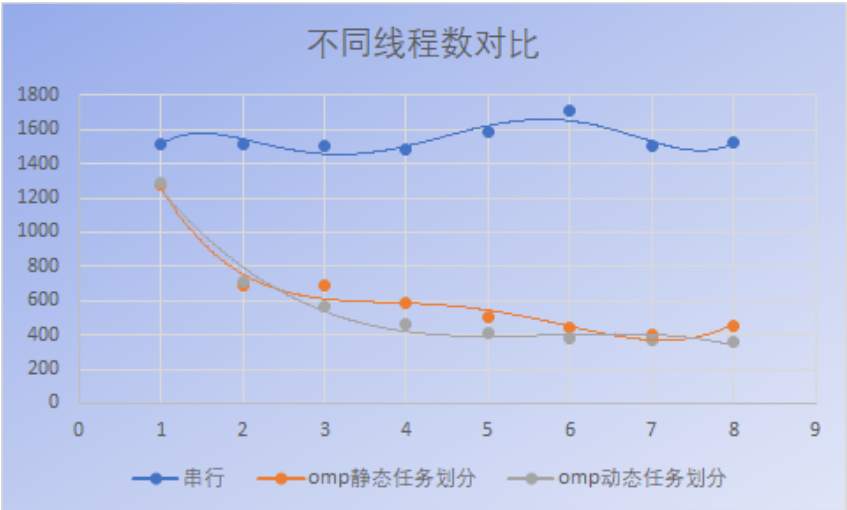


此结果是设置子线程数为 6 时得到的，上图和上表中可以看到整体上大致符合预期，和 ARM 平台的整体规律基本一样，但是这个的优化效果明显没有 ARM 的好，这和 pthread 实验得出的结论一

致，在线程开销上，X86 平台的线程开销更大，但是在 X86 平台上的动态任务划分有明显的优化效果，他比动态划分效果会更好一点。

3.3 不同线程数下结果分析

子线程数	串行/ms	静态划分/ms	动态划分/ms
8	1529.63	456.607	359.141
7	1500.74	403.305	373.077
6	1705.42	437.907	377.631
5	1586.69	508.946	408.258
4	1480.4	582.118	458.142
3	1504.28	691.396	566.209
2	1517.31	685.245	708.122
1	1519.71	1281.28	1287.3

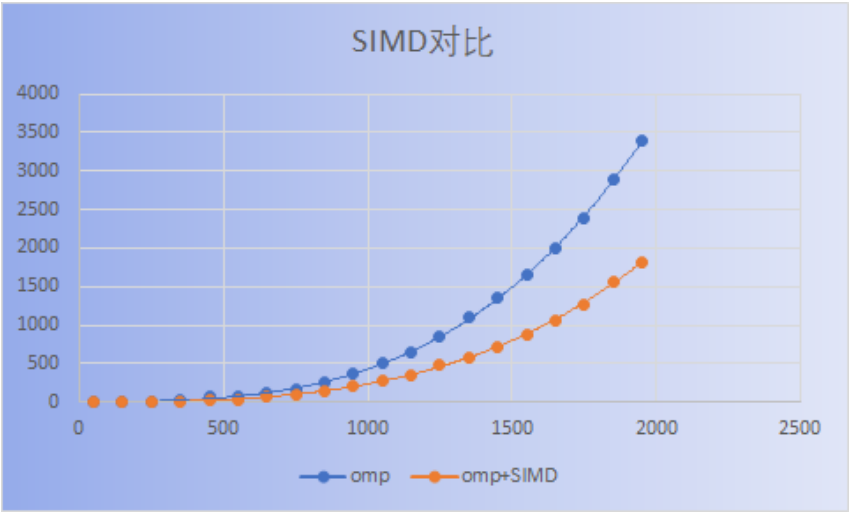


基于上述实验，这里也是选择了矩阵规模为 1000 进行实验测试，可以看到整体呈现出来的规律和 ARM 平台的规律是一致的，只是可能未对时间进行多次求平均，倒是串行时间曲线不是很平滑，且随着线程数的减小，运行时间也在不断增长，且在线程数较大时，这种减小不是很明显，但是在线程数为 1 时，omp 函数依旧有一定的优化。

3.4 Pthread+SIMD

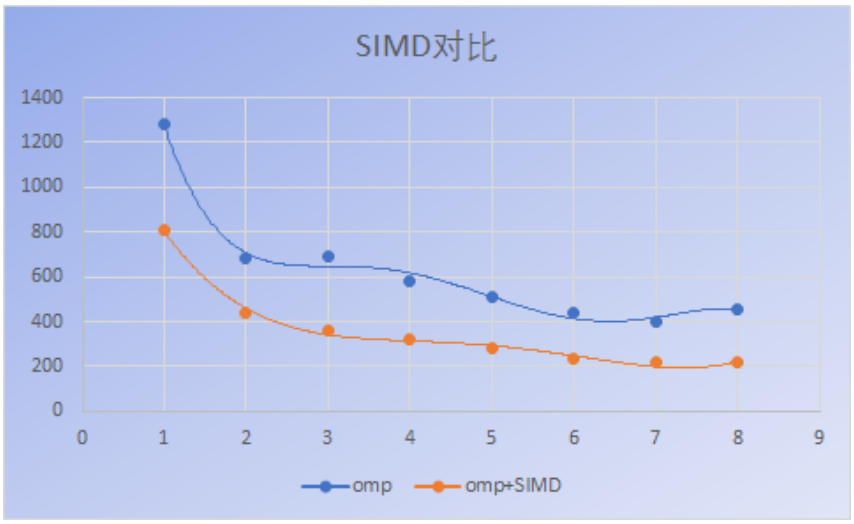
源码 github 链接戳这里。

数据规模	静态划分/ms	静态划分 +simd/ms
50	2.5984	0.1699
150	3.0979	1.3217
250	8.6661	6.4157
350	30.4849	14.2285
450	64.3559	37.559
550	70.7068	38.0563
650	112.853	62.9437
750	174.596	99.2652
850	259.009	144.514
950	363.808	200.443
1050	506.568	268.74
1150	645.787	353.234
1250	845.266	481.415
1350	1094.85	577.408
1450	1351.8	724.678
1550	1651.57	886.362
1650	2007.88	1065.89
1750	2382.1	1269.58
1850	2888.29	1567.26
1950	3387.64	1811.28



可以看到加入了 SIMD，整体效率的变化规律和未加入之前基本一样，对比上述的表格可以发现 SIMD 有一定程度上的优化，但是优化效果并不如只进行 SIMD 的效果好，这和 ARM 平台保持一致. 下面是不同线程时进行的 SIMD 进行的对比。

子线程数	omp/ms	omp+SIMD/ms
8	456.607	217.144
7	403.305	217.321
6	437.907	236.211
5	508.946	279.333
4	582.118	322.182
3	691.396	362.23
2	685.245	437.041
1	1281.28	812.197



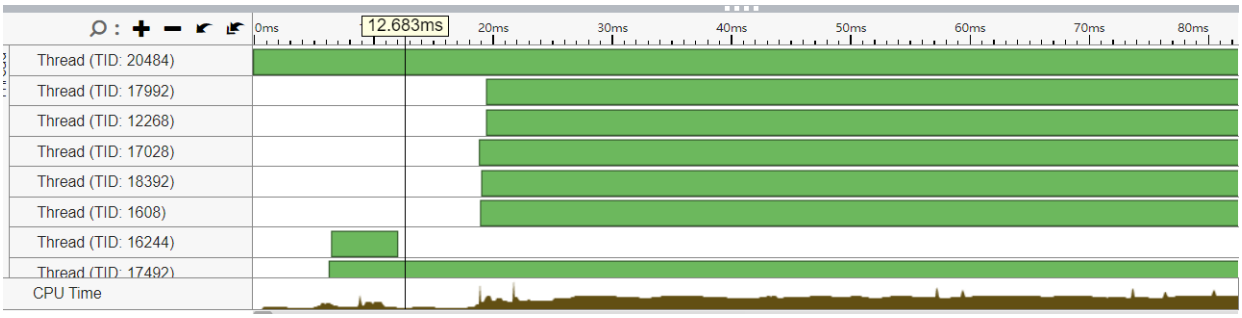
3.5 VTune 性能剖析

在 VTune 中进行性能分析，得到结果如下图所示

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Grouping: Function / Thread / Logical Core / Call Stack

Function / Thread / Logical Core / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
ompomp1	4.796s	14,418,300,000	20,703,800,000	0.696	48
Thread (TID: 20484)	0.835s	2,392,000,000	3,577,600,000	0.669	38
Thread (TID: 1608)	0.830s	2,486,900,000	3,499,600,000	0.711	53
Thread (TID: 12268)	0.824s	2,494,700,000	3,412,500,000	0.731	52
Thread (TID: 18392)	0.790s	2,410,200,000	3,373,500,000	0.714	56
Thread (TID: 17028)	0.775s	2,338,700,000	3,380,000,000	0.692	40
Thread (TID: 17992)	0.742s	2,295,800,000	3,460,600,000	0.663	47
ompdymanicomp1	4.683s	13,809,900,000	20,744,100,000	0.666	44
Thread (TID: 17992)	0.787s	2,325,700,000	3,149,900,000	0.738	55
Thread (TID: 20484)	0.785s	2,291,900,000	3,560,700,000	0.644	42
Thread (TID: 17028)	0.782s	2,325,700,000	3,499,600,000	0.665	56
Thread (TID: 1608)	0.780s	2,249,000,000	3,477,500,000	0.647	38
Thread (TID: 18392)	0.777s	2,273,700,000	3,707,600,000	0.613	29
Thread (TID: 12268)	0.773s	2,343,900,000	3,348,800,000	0.700	47
m_reset	3.832s	12,231,700,000	27,404,000,000	0.446	44
chuanxing	3.168s	10,582,000,000	24,077,300,000	0.440	44
Thread (TID: 20484)	3.168s	10,582,000,000	24,077,300,000	0.440	44
ompsimdomp1	2.507s	7,241,000,000	9,003,800,000	0.804	42
Thread (TID: 1608)	0.444s	1,306,500,000	1,523,600,000	0.858	40
Thread (TID: 20484)	0.424s	1,214,200,000	1,549,600,000	0.784	64
Thread (TID: 18392)	0.418s	1,194,700,000	1,496,300,000	0.798	43
Thread (TID: 17992)	0.418s	1,194,700,000	1,505,400,000	0.794	36
Thread (TID: 17028)	0.405s	1,181,700,000	1,443,000,000	0.819	37
Thread (TID: 12268)	0.398s	1,149,200,000	1,485,900,000	0.773	31



可以看到 omp 函数成功开辟出了 6 个线程，且对于每个线程上的时间 omp+SIMD< 静态任务划分 < 动态任务划分，这也再次证明了之前实验的结果，从各个线程工作的时间来看 20484 为主线程，他一直在工作，后来开辟出的线程被这几个函数共用。

4 总结

通过本次实验，更加熟悉了 OpenMP，也对多线程有了更深刻的认识。这次实验不用像上次一样自己进行任务划分，从 VTune 分析也可以看出，OpenMP 创建的是静态线程，他自己内部也界定了同步机制，这样操作就方便很多，但是对于线程数为 1 的优化，我目前还是存疑的，以后可能会进行进一步的学习。