南開大學
Nankai University

计算机学院

并行程序设计实验报告

自选题目：基于 COO 格式的稀疏矩阵乘法 pThread+ 优化

姓名：孟笑朵

学号：2010349

专业：计算机科学与技术

2022 年 5 月 6 日

Abstract

在本次实验中，利用常规的稀疏矩阵存储方式 COO 进行存储，实现了多种稀疏矩阵乘法 SpMM 和稀疏矩阵向量乘法 SpMV 的 pThread 优化并行算法，并结合 SIMD 算法进行进一步优化，分别实现 x86 平台，本机 Liunx 平台和鲲鹏 arm 平台的并行化加速，并利用性能测试与分析工具分析其相比于平凡算法的性能提升，给出相应的解释。此外，还探讨了同一程序多次进行动态分配，后几次动态分配的执行时间显著下降的问题，给出了分析与可能的原因。

关键字： SpMV，SpMM，pThread，SIMD

目录

1 性能测试环境

编程语言：C++
编译器：TDM-GCC
Windows10 操作系统：AMD CPU 笔记本

- 中心处理器：AMD Ryzen 7 5800H
- 硬盘：512GB
- 内存：16GB

华为鲲鹏服务器:

- 中心处理器: Linux master 4.14.0-115.el7a.0.1.aarch64
- 内存：195907MB

其中每个共有 96 个 CPU，每个 CPU 支持一个线程，一个 CPU 有 48 个核
Linux 环境：本机 WSL 环境

- 环境：Unbuntu 20.04 Linux 发布版
- 编译器：TDM-GCC

2 自选题介绍

本文是作为同杨鑫合作的《WMD 算法的并行化优化》的一个子问题的解决方案，该问题聚焦于 WMD 算法中最终求解的 Sinkhorn-Knopp 算法的并行优化，算法中涉及到多次稀疏矩阵与稠密向量矩阵相乘，因此，实现稀疏矩阵乘法的并行化十分有必要。不仅在本算法中，稀疏矩阵/张量在科学计算、数据分析、机器学习等应用中也十分常见，而稀疏矩阵的间接内存访问模式又给代码优化带来了巨大挑战，实现一种高效的稀疏矩阵乘法算法意义重大。

在《基于 COO 格式的稀疏矩阵乘法的 SIMD 编程》中，我们已经详细阐述了关于稀疏矩阵向量乘法的一些概念，并对稀疏矩阵压缩格式进行简单的介绍。**上次实验结束后，我们对比了稀疏矩阵 COO 格式存储与稀疏矩阵 CSR 格式存储的性能，发现 COO 格式存储格式稀疏矩阵乘法要比 CSR 存储格式的稀疏矩阵乘法慢 5-10 倍**! 这是难以接受的性能差距，这与 COO 格式的稀疏矩阵表示方法本身的性质不无关系，在本次实验中我们尝试使用 pThread 多线程对 COO 格式的稀疏矩阵乘法进行进一步的优化，同时探索更为高效的稀疏矩阵乘法存储格式，经过资料查阅，我们找到了一种较为高效的稀疏矩阵乘法格式——**HYB(HYBrid)。在 GPU 上，HYB(HYBrid) 是一种具有较高性能 SpMV 性能的稀疏矩阵存储格式，它使用 ELL(ELLPACK) 和 COO(COOrdinate) 两种存储格式来存储稀疏矩阵中非零元的信息**。

在本次实验中我们暂时继续讨论 COO 格式表示的稀疏矩阵乘法，为之后进行 HYB 格式的稀疏矩阵乘法做准备。本实验中继续将稀疏矩阵乘法划分为两类：一类是稀疏矩阵稠密向量相乘（SpMV），一类是稀疏矩阵稠密矩阵相乘（SpMM），将这两类问题分别实现 x86 平台，本机 linux 平台和鲲鹏 arm 平台的 SIMD 并行化加速，并比较分析其相比于平凡算法的性能提升。

3 并行算法实现与性能分析

3.1 SpMV 串行算法与并行分析

基于 COO 的稀疏矩阵格式与稠密矩阵相乘的算法伪代码如下：

$$\text{for } l = 0 \text{ to } k-1 \text{ do}$$
$$y[\text{ rowind }[l]] \leftarrow \text{y[rowind }[l]] + \text{val}[l] \cdot \text{x[colind}[l]]$$

在 SIMD 中，我们对这种算法进行了访存分析，发现这种算法至少要进行五次 memory load 才能进行一次运算，虽然我们也进行了循环展开的 SIMD 编程，但是由于这种算法本身的局限性，即空间访问并不具备连续性，强行进行 SIMD 的向量化反而适得其反，使得 SIMD 并行化算法反而比平凡算法性能更差。

对 COO 格式而言，稀疏矩阵的行与行在三个数组中的分界线不明显，导致我们无法利用 SpMV 算法的这种并行性，在上述算法中，实际上我们只能使用 1 个线程来进行计算，若强制并行计算，可能会造成对 y 的读写冲突，因此只能串行执行。

其中一种并行的思路是：使用多个线程同时计算稀疏矩阵中的若干个元素与 x 中相应元素的乘积，考察这些元素所在的行之间的关系，将得到的部分积累加到正确的 y 中元素上去，使用同步技术来解决访问冲突问题。

经过实验发现，这并不容易实现，而且对于上述简单的代码而言，实现过程显得冗余🅴嗦，我们期待更加自然的 pThread 并行算法。由此，我们提出了另一种并行思路：分析发现阻止 COO 格式的稀疏矩阵的乘法进行并行化的地方在于 COO 格式没有对行明确的界限，如果能够有一个数组专门对行的下标进行存储的话，我们就可以很容易地实现并行化。

基于上面的分析思路，改进了稀疏矩阵的 COO 存储格式如下：

COO 存储改进格式

```
class COOMatrix
{
    float* value,//存储非零元素的值
    int* row,//存储非零元素的行下标
    int* col,//存储非零元素的列下标
    int* index//存储每行第一个非零元素在row中的下标
};
```

3.2 不同平台下 SpMV 的 pThread 并行化算法

结合我们改进的 COO 稀疏矩阵格式，可以对上述串行的 SpMV 算法进行静态线程分配，下面是一种简单的块划分算法实现，算法伪代码如下：

```
void* coo_multiply_vector_pthread1(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int seg=nozerorows/THREAD_NUM;
    for(int i=index[seg*id];i<index[seg*(id+1)];i++){
```

```
            yy[row[i]] += value[i] * vec[col[i]];
        }
        pthread_exit(nullptr);
    }
```

在主线程内完成线程的创建销毁工作，具体的计算任务按行平均分配到每一个线程中，即每一个线程承担的行数是相同的。

另外，可以发现，**不同行中的非零元素的个数都是不同的，如果简单的进行静态的块划分，可能会造成线程任务分配不均的问题，针对这一问题，我们可以进行进一步的改进优化，将上述的静态线程改变为动态的线程分配模式**，我们进行了一些测试，可以发现的确存在这个问题：

```
thread0 6.14e-05
thread2 6.85e-05
thread3 7.24e-05
thread1 2e-07
```

动态线程的算法伪代码如下：

```
int next_arr2 = 0;
pthread_mutex_t  mutex_task;
void* coo_multiply_vector_pthread2(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int task = 0;

    while(1){
        pthread_mutex_lock(&mutex_task);
        task = next_arr++;
        pthread_mutex_unlock(&mutex_task);
        if (task >= nozerorows) break;
        for(int i=index[task];i<index[task+1];i++){
            yyy[row[i]] += value[i] * vec[col[i]];
        }
    }
    pthread_exit(NULL);
}
```

在线程数目为 4 时，三种算法在不同平台下不同规模下的性能测试如下：

针对上述两种线程分配模式，我们在 x86 平台下进行了进一步的性能测试比较如下图所示，数据规模是 100——10000 规模，线程数目为 4——12，所得到的结果如下：

可以发现，基于 pThread 的并行化优化随着数据规模的增大，性能也逐步增加，最终的性能比大概范围为 2 4 之间，最优的性能比已经达到了 3.2。下面我们来进一步比较静态线程和动态线程执行时间上的差异：

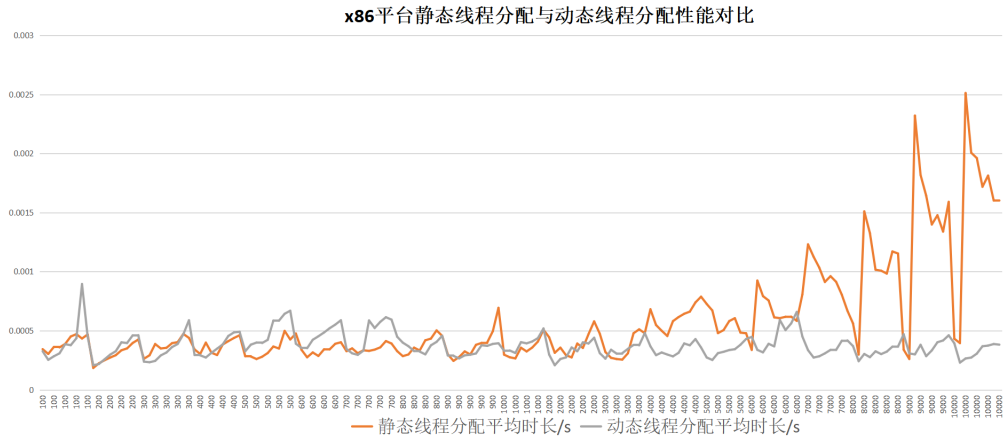

图 3.1: 在 x86 平台静态线程分配与动态线程分配的对比分析

从上述结果中可以看到，当矩阵规模较小（<4000）时，静态线程整体要比动态线性分配执行时间更少，但是当矩阵规模较大时（>4000），静态线程分配的执行时间变化幅度较大，而动态线程分配整体变化浮动更小，说明各个线程的分配更加均衡。

3.3 SpMV 的 pThread 并行化算法的性能测试

我们发现在数据规模较小时，线程数目为 4 的时候并行化的效果最佳，而随着线程数目的逐渐增多，开启线程销毁线程所占的时间小于算法本身的执行效率时，随着线程数目的增多，算法的性能得到进一步地提升，执行时间变短。下面进行更加细致的 perf 和 vTune 工具分析：

由于 SpMV 算法的 SIMD 并行化效果并不佳，在本节中没有对 pThread 和 SIMD 并行化进行结合，下面主要来看一下 SpMM 算法的并行化优化：

3.4 SpMM 串行算法与并行分析

在 SIMD 实验中，我们进行简单的 Cache 优化所得到稀疏矩阵相乘算法伪代码如下：

```
1  void coo_multiply_matrix_serial(){
2      for (int i=0;i<nonzeros;i++)
3          for(int k=0;k<n;k++)
4              c[row[i]][k] += value[i] * b[col[i]][k];
5  }
```

在 SIMD 编程中我们对内层循环进行了循环展开操作，得到了 1.5 2 倍左右的加速比，在 pThread 编程中，我们有两种编程的思路，一种是对内层循环进行线程分配，一种是对外层循环进行线程分配。下面是对两种并行化思路分别进行实现并进行性能对比分析：

对内层循环进行线程分配就是在主线程内部进行外层循环和线程划分，子线程进行计算内部循环，伪代码如下：

```
1  //线程函数
2  void* coo_multiply_matrix_pthread2(void *parm){
```

```
    threadParm_t2 *p = (threadParm_t2 *) parm;
    int id = p->threadId;
    int i=p->rowid;
    int interval=n/THREAD_NUM;
    int maxx=0;
    if(id==3){
        maxx=n;

    }else{
        maxx=interval*(id+1);
    }
    for(int k=interval*id;k<maxx;k++)
        mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
    pthread_exit(NULL);
}

//主线程函数
void spMM_pThread_static2(int thread_num){
    THREAD_NUM=thread_num;
    pthread_t thread[THREAD_NUM];
    threadParm_t2 threadParm[THREAD_NUM];
    for (int j=0;j<nonzeros;j++)
    {
        for (int i = 0; i < THREAD_NUM; i++)
        {
            threadParm[i].threadId = i;
            threadParm[i].rowid = j;
            pthread_create(&thread[i], nullptr, coo_multiply_matrix_pthread2,
            (void *)&threadParm[i]);
        }
        for (int i = 0; i < THREAD_NUM; i++)
        {
            pthread_join(thread[i], nullptr);
        }
    }
}
```

对外层循环进行线程分配的思路与 SpMV 算法的并行化思路相同，都是对不同线程进行行的划分，进行计算求解，算法思路如下：

```
//子线程函数
void* coo_multiply_matrix_pthread1(void *parm){
```

```
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int interval=nozerorows/THREAD_NUM;
    int maxx=0;
    if(id==3){
        maxx=nonzeros;
    }else{
        maxx=index[interval*(id+1)];
    }

    for(int i=index[interval*id];i<maxx;i++){
        for(int k=0;k<n;k++)
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
    }
    pthread_exit(NULL);
}

//主线程
void spMM_pThread_static1(int thread_num){
    THREAD_NUM=thread_num;
    pthread_t thread[THREAD_NUM];
    threadParm_t threadParm[THREAD_NUM];
    for (int i = 0; i < THREAD_NUM; i++)
    {
      threadParm[i].threadId = i;
      pthread_create(&thread[i], nullptr, coo_multiply_matrix_pthread1,
      (void   *)&threadParm[i]);
    }

    for (int i = 0; i < THREAD_NUM; i++)
    {
      pthread_join(thread[i], nullptr);
    }
}
```

在这里，我们简单地对比两种算法的性能，以数据规模为 4096，稀疏度为 0.002 为例进行性能分析，得到下面表格中的数据：

可以发现对内层循环的 pThread 并行化优化取得的效果反而比之前的平凡算法还要差，跟与外层循环相比，相差的倍数甚至是数量级的，通过 perf 分析，我们可以看出第一种算法的 cache 的 miss 次数明显大于平凡算法和第二种 pThread 算法，另外，可以发现在主线程中进行了多次的线程创建与销毁工作，其中造成的开销远远超过算法本身的计算量。

3.5 动态线程分配 pThread 并行化算法

同样的上述的静态线程分配也存在一个问题，就是可能遇到行中非零元素的个数不一样，造成各个线程的任务分配不均的问题，同样实现了线程的动态分配，算法的伪代码如下：

```
int next_arr2 = 0;
pthread_mutex_t  mutex_task;
void* coo_multiply_matrix_pthread3(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int task = 0;
    int maxx;
    while(1){
        pthread_mutex_lock(&mutex_task);
        task = next_arr2++;
        pthread_mutex_unlock(&mutex_task);
        if (task >= nozerorows) break;
        if(task>=nozerorows-1)maxx=nonzeros;
        else maxx=index[task+1];
        for(int i=index[task];i<maxx;i++){
            for(int k=0;k<n;k++)
                mat_res3[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    pthread_exit(NULL);
}
```

在这种动态线程分配中，子线程每完成一行就进行一次任务分配，继续分配一行的任务，每次以一行的范围进行划分，划分粒度过细，我们改进了我们动态分配线程方法，将任务划分的粒度进行变量控制，得到了下面这种算法设计，代码如下：

```
int next_arr2 = 0;
int single_circle=50;
pthread_mutex_t  mutex_task;
void* coo_multiply_matrix_pthread3(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int task = 0;
    int maxx;
    while(1){
        pthread_mutex_lock(&mutex_task);
        task = next_arr2;
```

单位线程任务	平凡算法	静态分配 pThread	动态线程分配 pThread	动态分配加速比
20	0.236941	0.088937	0.0803853	2.94757
30	0.239057	0.0984004	0.0786337	3.04014
40	0.239909	0.0790709	0.077028	3.11458
50	0.23557	0.0817924	0.083394	2.82478
60	0.240548	0.0811911	0.0751527	3.20078
70	0.239274	0.0776491	0.0779992	3.06764
80	0.243582	0.0934441	0.0779238	3.1259
90	0.242993	0.11325	0.0817277	2.9732

```
            next_arr2+=single_circle;
            pthread_mutex_unlock(&mutex_task);
            if (task >= nozerorows) break;
            if(task>=nozerorows-single_circle)maxx=nonzeros;
            else maxx=index[task+single_circle];
            for(int i=index[task];i<maxx;i++){
                for(int k=0;k<n;k++)
                    mat_res3[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
            }
        }
        pthread_exit(NULL);
    }
```

3.5.1 对动态分配 pThread 优化中的单位任务数量进行探讨

基于以上算法，我们针对矩阵规模为 4096, 稀疏度为 0.002 的稀疏矩阵和稠密矩阵进行乘法操作，在不同的动态分配的单位任务数量下进行探讨，其中单位任务的范围为 10——100，代表每一次动态规划所得到的结果，如下表所示：

可以看到，动态分配优于静态分配的效果，对于平凡算法的加速比也达到了 2.9——3.2 的加速效果，是较为明显的提升，在不同的单位线程任务中，可以发现，在 50 附近时的算法的加速比最大，算法的性能最好。

3.6 SpMM 算法 pThread+SIMD 并行化算法

进一步地，我们看到我们只对外层循环进行了任务划分，而并没有对内层循环进行任务划分，对于内层循环，正好容易利用内层循环 cache 空间访问的特性对内层循环进行 SIMD 并行化优化，并行化优化的思路有如下几种：

1. pthread 静态线程分配 +sse 并行化
2. pthread 静态线程分配 +avx 并行化
3. pthread 动态线程分配 +sse 并行化
4. pthread 动态线程分配 +avx 并行化
5. pthread 静态线程分配 +neno 并行化
6. pthread 动态线程分配 +neno 并行化

这里展示 pthread 静态线程分配 +sse 并行化算法，与 pthread 动态线程 +neon 伪代码为例，其余算法实现均类似。

```
//pthread 静态线程分配 +sse 并行化
void* coo_multiply_matrix_pthread_sse1(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int id = p->threadId;
    int interval=nozerorows/THREAD_NUM;
    int maxx=0;
    __m128 t1,t2,t3,sum;
    int choice = n % 4;
    if(id==3){
        maxx=nonzeros;

    }else{
        maxx=index[interval*(id+1)];
    }

    for(int i=index[interval*id];i<maxx;i++){
        for(int k=0;k<n-choice;k+=4)
        {
            t1=_mm_load_ps(mat_nonsparse[col[i]]+k);
            sum = _mm_setzero_ps();
            t3 = _mm_set_ps1(value[i]);
            t2=_mm_load_ps(mat_res1[row[i]]+k);
            sum = _mm_mul_ps(t3,t1);
            t2=_mm_add_ps(t2,sum);
            _mm_store_ps(mat_res1[row[i]]+k,t2);
        }
        for(int k=n-choice;k < n;k++){
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
    pthread_exit(NULL);
}
```

```
//pthread 动态线程分配 +neno 并行化
double spMM_pThread_dynamic_neon(int thread_num){
    THREAD_NUM=thread_num;
    struct timeval val,newVal;
    gettimeofday(&val, NULL);
```

```
6       pthread_t thread[THREAD_NUM];
7       threadParm_t threadParm[THREAD_NUM];
8
9       for (int i = 0; i < THREAD_NUM; i++)
10      {
11        threadParm[i].threadId = i;
12        pthread_create(&thread[i], nullptr, coo_multiply_matrix_pthread4, (void
13            *)&threadParm[i]);
14      }
15
16      for (int i = 0; i < THREAD_NUM; i++)
17      {
18        pthread_join(thread[i], nullptr);
19      }
20      gettimeofday(&newVal, NULL);
21      unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec) -
22      (val.tv_sec     * Converter + val.tv_usec);
23 I    return diff / Converter;
24  }
```

3.7 SpMM 并行算法性能分析

如图为 x86 平台下不同算法的性能差异，但是分析该图，我们发现，**对于动态分配线程，当我们循环改变了矩阵的稀疏度时，由于某种原因，动态分配的 pThread 的程序只在第一次得到较为“正常”的值，在这之后得到的值都明显不正常**，可以查看表格发现，这里以矩阵规模为 9000,10000, 稀疏度为 0.01——0.04 为例进行说明：

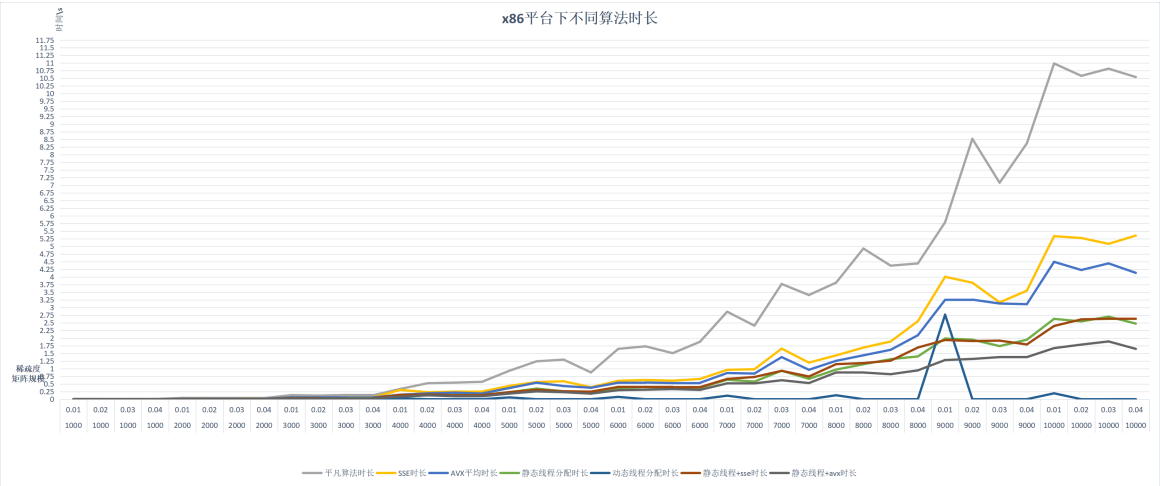

图 3.2: 在 x86 平台 spmm 并行算法对比分析

初步猜测是因为在首次创建线程，线程任务结束之后，并没有直接结束线程，在下一次进行计算时，直接“唤醒”线程，从而降低了线程创建时的开销，达到下次进行运算时“拿来就用”的效果；当

矩阵规模	稀疏度	动态线程分配时长
9000	0.01	2.7664
9000	0.02	0.0005656
9000	0.03	0.0016844
9000	0.04	0.0010432
10000	0.01	0.193248
10000	0.02	0.0003981
10000	0.03	0.0011133
10000	0.04	0.0011551

然也有可能是编译器在其中起到了一定的作用，将程序性能更加优化了；由于时间缘故，与笔者笔记本的限制原因，本应该进一步开展 vTune 线程分析工作，受条件所限，笔者只能给出造成这种情况的一个猜测。

perf 工具和火焰图分析如下

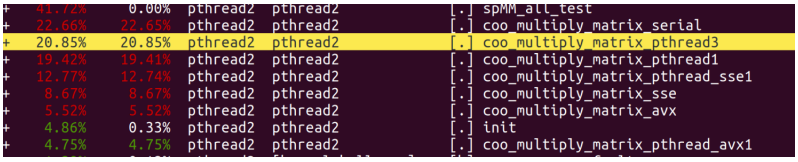

图 3.3: 在 arm 平台 spmm 并行算法 perf 分析

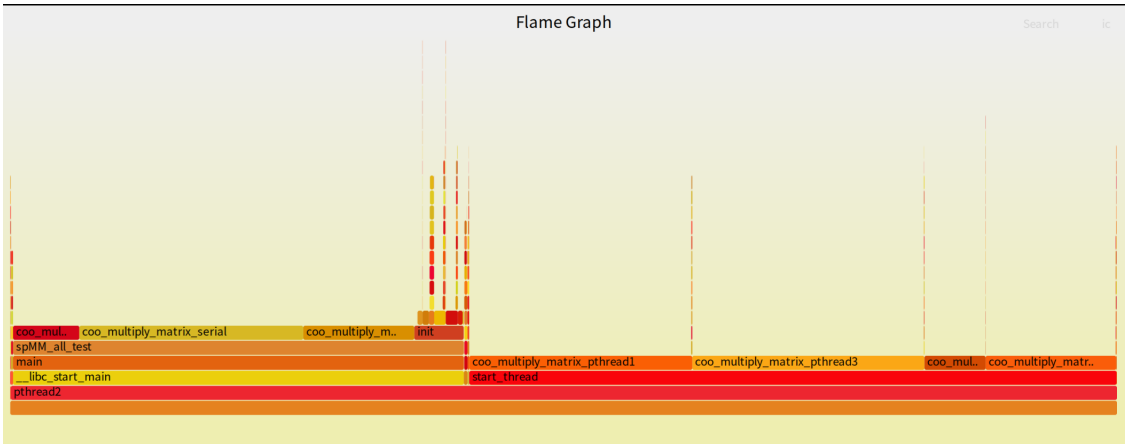

图 3.4: 在 arm 平台 spmm 并行算法火焰图分析

4 代码上传

全部代码素材已上传GitHub.