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 openMP+ 优化

姓名：孟笑朵

学号：2010349

专业：计算机科学与技术

2022 年 5 月 21 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 openMP 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。

关键字： SpMV，SpMM，openMP，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:

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

测试得到本机一共可以开启 16 个线程
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

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。在《基于 COO 格式的稀疏矩阵乘法的 pThread 编程》中，我们进行了稀疏矩阵向量乘的 pThread 并行优化并进行了性能测试，并对 HYB(HYBrid) 格式表示的稀疏矩阵进行了简单介绍。**上次实验结束后，发现上次实验中第一次动态线程分配后的程序运行效率显著的原因在于，指向资源开始的指针变量在第一动态线程分配之后没有初始化，造成了不正常的性能变化**。在本次实验中，我们继续使用 COO 格式的稀疏矩阵乘法的并行化优化，在 GPU 并行化中使用 HYB 格式的稀疏矩阵乘法并行化优化，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：

for $l = 0$ to $k - 1$ 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。在 pThread 编程中，我们优化了 COO 的稀疏矩阵表示格式，并说明了对外层的多线程优化要明显优于对内层的多线程并行化。

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 openMP 并行化算法

优于 openMP 是基于串行程序，然后添加代码，然后 openMP 自动为我们提供并行化优化，所以我们需要分析程序，基于改进的存储格式设计串行算法，算法伪代码如下：

```
int i,j;
for(i=0;i<nozerorows;i++)
{
    for(j=index[i];j<index[i+1];j++)
    {
        yy[row[j]]+=value[j]*vec[col[j]];
    }
}
```

在这种算法的基础上进行 openMP 编程，首先分析程序的依赖关系，其中外层是一个可以进行并行划分的部分内层由于 yy[row[j]] 写入数据造成的数据冲突，不太好实现并行化，而且在 pthread 编程中，我们分析得到内层进行并行化会破坏程序的 cache 优化，效率要明显外层的并行化效率。

```
int i,j;
#pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, j)
#pragma omp for
for(i=0;i<nozerorows;i++)
{
    for(j=index[i];j<index[i+1];j++)
```

```
    {
        yy[row[j]]+=value[j]*vec[col[j]];
    }
}
```

openMP 为我们提供了很方便的 API，使得我们很容易进行静态动态线程分配，这里实现 guided 动态调整的动态线程分配算法：

```
int i,j;
#pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, j)
//#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
#pragma omp for schedule(guided)
for(i=0;i<nozerorows;i++)
{
    for(j=index[i];j<index[i+1];j++)
    {
        yy[row[j]]+=value[j]*vec[col[j]];
    }
}
```

3.3 SpMV 的 pThread 并行化算法的性能测试

我们发现对比 pthread 来讲 openMP 的优化效果更好，在数据规模较小的情况下，多线程的优化效果并不佳，而 openMP 的优化效果要明显优于平凡算法和 pThread 优化算法；而且相对于平凡算法，openMP 算法的程序的运行效率可以达到 10 倍左右的加速比。

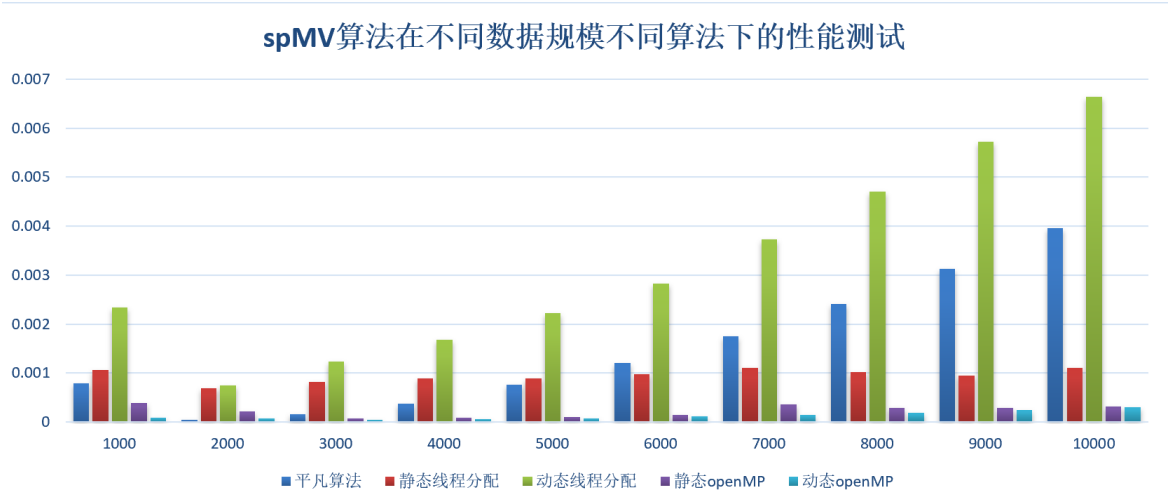

图 3.1: 在 arm 平台 spMV 算法在不同数据规模不同算法下的性能测试

由于 SpMV 算法的 SIMD 并行化效果并不佳，在本节中没有对 openMP 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：

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

对于稀疏矩阵乘法，我们采取的 openMP 并行化思路和 pThread 中并行化思路类似，在稀疏矩阵乘法的外层循环进行多线程循环划分，在稀疏矩阵乘法的内层进行 SIMD 向量并行化。具体编程如下：

```
void coo_multiply_matrix_openMP_static(){
    int i,k;
    #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i, k)
    for (i=0;i<nonzeros;i++)
    {
        for(k=0;k<n;k++)
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
}
```

3.5 动态线程分配 openMP 并行化算法

对比 pThread 编程，openMP 可以使用 schedule 来确定线程划分的方式，和每个线程划分的任务数目，对于动态 openMP 编程提供了 guided 参数用于动态调整线程数目分配，对比 pThread 编程更加方便

```
void coo_multiply_matrix_openMP_dynamic(){
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, k)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for (i=0;i<nonzeros;i++)
    {
        for(k=0;k<n;k++)
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
    }
}
```

3.6 SpMM 算法 openMP+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. openMP 静态线程分配 +sse 并行化
2. openMP 静态线程分配 +avx 并行化
3. openMP 动态线程分配 +sse 并行化
4. openMP 动态线程分配 +avx 并行化
5. openMP 静态线程分配 +neno 并行化
6. openMP 动态线程分配 +neno 并行化

```
///实现 spMM 的 openMP 编程版本静态线程分配 +SSE 并行
void coo_multiply_matrix_openMP_static_sse(){
    __m128 t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i, k,t1,t2,t3,sum)
    for (i=0;i<nonzeros;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];
        }
    }
}

///实现 spMM 的 openMP 编程版本动态线程分配 +SSE 并行
void coo_multiply_matrix_openMP_dynamic_sse(){
    __m128 t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i, k,t1,t2,t3,sum)
```

```
31      //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
32      #pragma omp for schedule(guided)
33      for (i=0;i<nonzeros;i++)
34      {
35          for(int k=0;k<n-choice;k+=4)
36              {
37                  t1=_mm_load_ps(mat_nonsparse[col[i]]+k);
38                  sum = _mm_setzero_ps();
39                  t3 = _mm_set_ps1(value[i]);
40                  t2=_mm_load_ps(mat_res1[row[i]]+k);
41                  sum = _mm_mul_ps(t3,t1);
42                  t2=_mm_add_ps(t2,sum);
43                  _mm_store_ps(mat_res1[row[i]]+k,t2);
44              }
45          for(int k=n-choice;k < n;k++){
46              mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
47          }
48      }
49  }
```

类似地，可以在 ARM 鲲鹏平台上实现结合 NEON 的并行化:

```
1   ///实现 spMM 的 openMP 编程版本静态线程分配 +NEON 并行
2   void coo_multiply_matrix_openMP_static_neon(){
3       float32x4_t t1,t2,t3,sum;
4       int choice = n % 4;
5       int i,k;
6       #pragma omp parallel for num_threads(OMP_NUM_THREADS),private(i,
7       k,t1,t2,t3,sum),shared(choice,mat_nonsparse,mat_res1,value,row,col)
8       for (i=0;i<nonzeros;i++)
9       {
10          for(int k=0;k<n-choice;k+=4)
11          {
12              sum=vdupq_n_f32(0.f);
13              t1=vld1q_f32(mat_nonsparse[col[i]]+k);
14              t2=vld1q_f32(mat_res1[row[i]]+k);
15              t3 = vdupq_n_f32(value[i]);
16              sum = vmulq_f32(t3,t1);
17              t2= vaddq_f32(t2,sum);
18              vst1q_f32(mat_res1[row[i]]+k,t2);
19          }
20          for(int k=n-choice;k < n;k++){
```

```
            mat_res1[row[i]][k] += value[i] * mat_nonsparse[col[i]][k];
        }
    }
}

///实现 spMM 的 openMP 编程版本动态线程分配 +NEON 并行
void coo_multiply_matrix_openMP_dynamic_neon(){
    float32x4_t t1,t2,t3,sum;
    int choice = n % 4;
    int i,k;
    #pragma omp parallel num_threads(OMP_NUM_THREADS),private(i,
    k,t1,t2,t3,sum),shared(choice,mat_nonsparse,mat_res1,value,col,row)
    //#pragma omp for schedule(static, nozerorows/OMP_NUM_THREADS)dynamic, 50
    #pragma omp for schedule(guided)
    for (i=0;i<nonzeros;i++)
    {
        for(int k=0;k<n-choice;k+=4)
        {
            sum=vdupq_n_f32(0.f);
            t1=vld1q_f32(mat_nonsparse[col[i]]+k);
            t2=vld1q_f32(mat_res1[row[i]]+k);
            t3 = vdupq_n_f32(value[i]);
            sum = vmulq_f32(t3,t1);
            t2= vaddq_f32(t2,sum);
            vst1q_f32(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];
        }
    }
}
```

3.7 SpMM 并行算法性能分析

3.7.1 不同线程数目对程序性能的影响

首先，我们在鲲鹏 arm 平台下分析了对于不同线程数目 openMP 程序性能的影响，设置线程数目为 10——100, 矩阵的规模为 4096，分析比较了静态线程，静态线程 +NEON 并行化，动态线程，动态线程 +NEON 并行化相对于平凡算法的加速比，如下图所示：

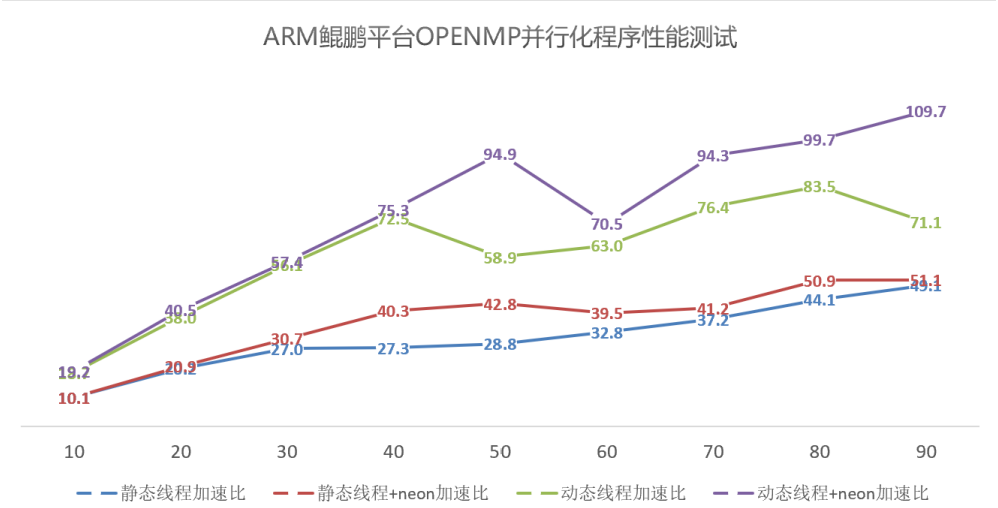

图 3.2: 不同线程数目下 ARM 鲲鹏平台 openMP 程序加速比测试

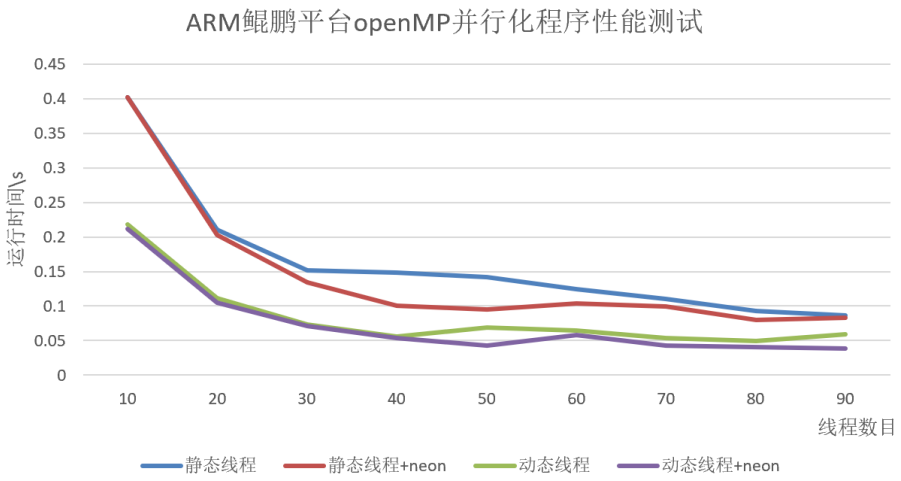

图 3.3: 不同线程数目下 ARM 鲲鹏平台 openMP 程序运行时间测试

分析测试图，我们发现，在 10—40 线程范围内，静态线程分配的加速比线程数目大致成正比，在线程数目较少时，动态线程分配的加速比明显高于线程数目增长的比例，在线程数目较多时，加速比大致与线程数目的增长比例相同，在 90 线程的情况下，动态线程 +NEON 的加速比甚至达到了 100 倍。

分析程序运行时间随着程序线程数目变化图，可以发现在 10—40 线程内程序性能的变化较为明显，在线程数目大于 40 时，程序性能的变化不太明显，有时会出现线程数目增加程序性能反而下降的情况，这是因为对于多线程而言，线程增加造成的开销要高于线程数目增加带来的程序的性能提升。

3.7.2 不同平台上的性能测试

我们分别在 x86, 本机 linux，Intel OneAPI 的 DEV 平台上进行了程序性能测试，测试了平凡算法，SIMD，pthread，openMP 几种算法的运行的性能，如下图所示：

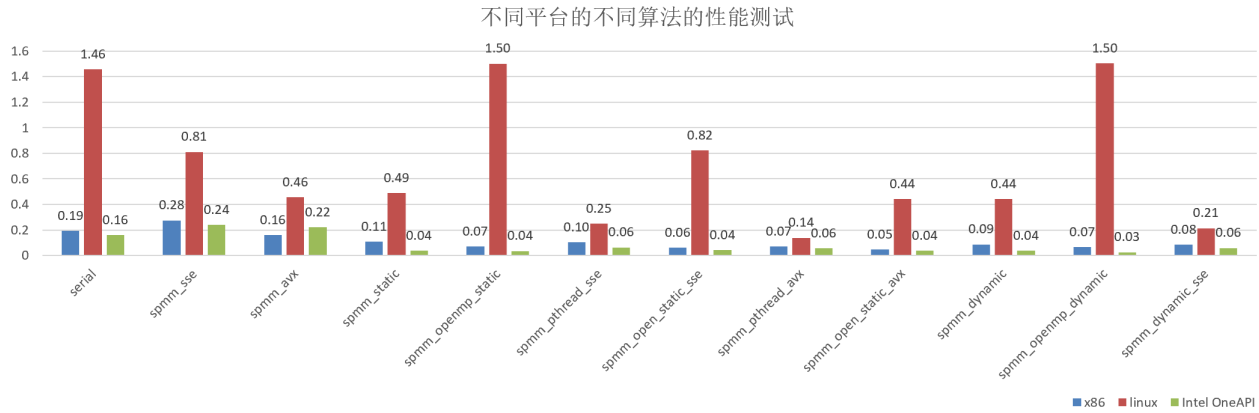

图 3.4: 不同平台上不同算法的程序性能分析图

可以发现对比 x86 平台的程序运行效率，Intel OneAPI 的程序运行效率要高 1—1.5 倍左右，相比于平凡算法的程序运行效率提升了 5 倍。

ARM 鲲鹏平台上的程序性能分析如下表所示，其中矩阵的规模为 4096,openMP 和 pthread 的线程数目均为 16.

算法	运行时间
serial	4.08947
spmm_neon	2.08836
spmm_static	3.17473
spmm_openmp_static	0.288023
spmm_dynamic	0.28378
spmm_openmp_dynamic	0.270902
spMM_static_neon	2.59854
spmm_open_static_neon	0.139566
spmm_dynamic_neon	0.244391
spmm_open_dyna_neon	0.131194

以上的程序运行是在未加入编译器的优化的情况下运行的，可以发现 openMP 的程序运行效率不论是动态的还是静态的都要明显优于 pThread 的程序运行效率，在 openMP 的基础上再加入 NEON 向量化的程序并行化，使得程序运行的效率达到最优，最优的程序运行效率为平凡算法的 30 倍左右。

perf 工具和火焰图分析如下

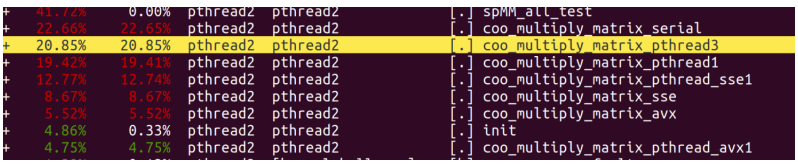

图 3.5: 在 arm 平台 spmm 并行算法 perf 分析

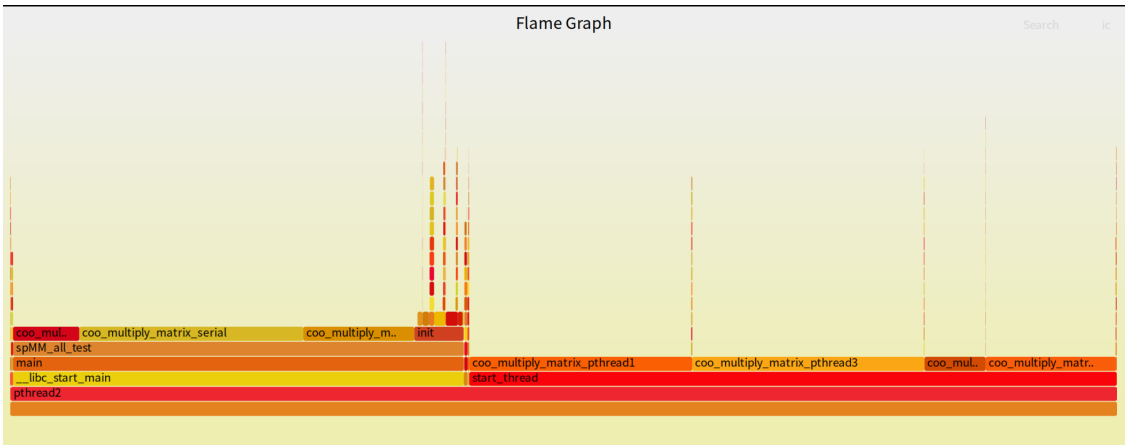

图 3.6: 在 arm 平台 spmm 并行算法火焰图分析

4 总结与收获

在本次实验中，得到了以下几点收获：

1. 学会了利用 Intel OneAPI 提供的在线 DEV cloud 进行高性能程序编译执行；

2. 学号了使用 gdb 命令对 g++ 编译的程序进行调试，在本次实验中，程序在数据规模较大时会出现段溢出（Segmentation fault (core dumped)）的问题，使用 ulimit -a 查看系统是否配置支持了 dump core 的功能，再利用 ulimit -c 1024 开启了 core 文件记录，再利用 gdb 调试器查看了详细的报错信息；另外，学习了在 linux 平台下利用 gdb 进行程序的调试，定位到了异常点，解决了一部分程序数据规模较大时的段溢出问题，但是发现在数据规模更大的情况下还是会发生溢出的问题，使得无法测试较大规模的程序的运行效率；

5 代码上传

全部代码素材已上传GitHub.