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 的稀疏矩阵乘法 SIMD 优化

姓名：孟笑朵

学号：2010349

专业：计算机科学与技术

2022 年 4 月 12 日

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

- 中心处理器：Unbuntu 20.04 Linux 发布版

2 自选题介绍

2.1 简介

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。下面简单阐述一些概念：

- 稀疏矩阵向量乘 (Sparse Matrix-Vector Multiplication，SpMV) 是科学计算领域一个非常重要的内核，在求解稀疏线性方程组的迭代法中占据了主要的计算量。
- 稀疏矩阵的元素大部分是零，非零元素所占比例往往小于元素总数的 1%，因此，稀疏矩阵的存储往往采用压缩格式，只存储非零元和非零元在矩阵中的位置。
- 将矩阵用稀疏的格式表示其实已经是一种优化了（相对稠密的格式），本文讨论的是对于使用稀疏格式的计算，还能如何继续优化，对于普通的稀疏矩阵稠密表示不做讨论。

本文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 平台和 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。

2.2 稀疏矩阵压缩格式

本文中稀疏矩阵的存储压缩格式为 COO（coordinate (COO) format）格式，小组中的另一名成员杨鑫使用的是 CSR 压缩格式，方便比较不同稀疏矩阵压缩格式在不同平台和不同参数下的性能。

COO 格式使用三个数组来存储一个稀疏矩阵中的非零元：

- row[nonzeros]: 存储每个非零元的行索引;
- col[nonzeros]: 存储每个非零元的列索引;
- value[nonzeros]: 存储每个非零元的值。

这种方法表示的优势在于实现简单，容易实现稀疏矩阵与 COO 表示的压缩格式互转，缺点在于三个数组是割裂开的，访问格式是间接的，难以实现 cache 缓存。

3 并行算法设计

3.1 SpMV 串行算法与并行分析

基于这种压缩格式，与稠密向量相乘的串行算法的伪代码如下：

1 **for** $l = 0$ **to** $k - 1$ **do**
2 $\quad \mathtt{y}[\mathtt{rowind}[l]] \leftarrow \mathtt{y}[\mathtt{rowind}[l]] + \mathtt{val}[l] \cdot \mathtt{x}[\mathtt{colind}[l]]$

图 3.1: SpMV 串行算法伪代码

该代码的具体访存分析如下：

每轮 l 需要 5 次 memory loads，rewind[l]，colind[l] 和 val[l] 的访问都具有 spatial 局部性，但是那俩对于 x 和 y 的间接访问则局部性取决于 runtime 的数据分布。如前文所述，矩阵元素都只会被访问一次，所以只有 x 和 y 的访问才可能重用。譬如如果连续的 rewind[l] 都返回一样的值，则 y 的访问存在 temporal 局部性；如果连续的 rewind[l] 都返回连续的值，则 y 的访问存在 spatial 局部性。

在这里，我们可以看到一个最基本的并行思路就是循环展开，基于这种并行思想，我们可以设计对应的并行算法：

3.1.1 x86 平台下 SSE 与 AVX 并行优化编程

可以将变量都加载到 SSE 提供的向量寄存器中，然后进行运算，算法的代码如下：

```
1  void coo_multiply_vector_sse1(int nonzeros,int n,int* row,
2  int* col,float* value,float* x,float* y){
3      __m128 t1,t3,sum;
4      int choice = nonzeros % 4;
5      for(int i=0;i<n;i++)
6          y[i]=0;
7      for (int i=0;i<nonzeros-choice;i+=4){
8          t1 = _mm_set_ps(x[col[i+3]],x[col[i+2]],x[col[i+1]],x[col[i]]);
```

```
            sum = _mm_setzero_ps();
            t3 = _mm_load_ps(value+i);
            sum = _mm_mul_ps(t3,t1);
            for(int j=0;j<4;j++)
                y[row[i+j]] += sum[j];
        }
        //对齐操作
        for(int i=nonzeros - choice; i < nonzeros ; i++){
            y[row[i]] += value[i] * x[col[i]];
        }
I}

```

后面我们会提到基于这种思想设计的算法实际上效率并不高，主要原因在于这些变量存放的地址并不连续，不连续的变量进行向量化需要将内存一个一个“加载”到向量寄存器中，增加了访存的次数，另外我们在内层循环嵌套了一个小循环，总的循环次数并没有增加，并不是一种好的并行思路。

3.1.2 ARM 平台下 neon 并行优化编程

同样的，也可以在 ARM 平台上借助 neon 来实现向量化，代码如下：

```
void coo_multiply_vector_neon(int nonzeros,int n,int* row,
int* col,float* value,float* x,float* y){
    int choice = nonzeros % 4;
    for(int i=0;i<n;i++)
        y[i]=0;
    for (int i=0;i<nonzeros-choice;i+=4){
        sum=vdupq_n_f32(0.f);
        t1=vsetq_lane_f32(x[col[i+3]],t1,3);
        t1=vsetq_lane_f32(x[col[i+2]],t1,2);
        t1=vsetq_lane_f32(x[col[i+1]],t1,1);
        t1=vsetq_lane_f32(x[col[i]],t1,0);
        //设置向量 v 第 lane 个通道的元
        t3 = vld1q_f32(value+i);
        sum = vmulq_f32(t3,t1);
        for(int j=0;j<4;j++)
            y[row[i+j]] += sum[j];
    }
    //对齐操作
    for(int i=nonzeros - choice; i < nonzeros ; i++){
        y[row[i]] += value[i] * x[col[i]];
    }
}
```

在后面的性能测试中，无论是在 x86 平台上还是在 ARM 平台上，得到的性能测试结果并不理想，几乎是原来串行算法的一倍。分析其原因，在于在 SIMD 向量化的过程中，本身进行的访存次数就超过了原来串行化的访存次数，非但没有起到并行优化的效果，反而降低了原来算法的效率。有待找到一种更加合适的并行化思路来实现。受笔者水平所限，暂时无法用 SIMD 来实现。

3.2 SpMM 串行算法与并行分析

这里我们有两种算法的思路，一种思路就是先将稠密矩阵进行转置，转置之后利用上面我们进行向量优化的思路进行稀疏矩阵稠密矩阵乘法的并行化优化，另一种思路是先进行 cache 优化再进行矩阵的乘法运算以及并行化的实现。

但是通过比较我们可以发现，第一种算法根据上面我们的分析，并行化对其的优化效果并不好，而且该算法的 cache 性能并不高，第二种算法是进行 cache 优化后的矩阵乘法思路，相对而言效率会更高，因此在这里我们选取第二种算法作为我们实现并行化的串行算法基础。串行算法代码如下：

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

```

很明显，我们能看到，对于以上的串行算法并行的一个基本的思路是将循环展开，进行向量化处理，基于此，我们设计了以下的并行编程算法。为了对比效果明显，我们将第一种思路及其并行化也进行实现了，下面展示两种算法并行化思路与理论分析：

3.2.1 x86 平台下 SSE 与 AVX 并行优化编程

第一种思路的 SSE 并行化如下：

```
void coo_multiply_matrix_sse1(int nonzeros, int n, int* row,
int* col, float**value, float** b, float** c) {
    int choice = nonzeros % 4;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;
    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < nonzeros - choice; i += 4)
```

```
        {
            t1 = _mm_set_ps(b[col[i + 3]][k], b[col[i + 2]][k],
            b[col[i + 1]][k], b[col[i]][k]);
            sum = _mm_setzero_ps();
            t3 = _mm_load_ps(value + i);
            sum = _mm_mul_ps(t3, t1);
            //加了这个循环使得并行的效果被掩盖了
            for (int j = 0; j < 4; j++)
                c[row[i + j]][k] += sum[j];
        }
        for (int i = nonzeros - choice; i < nonzeros; i++) {
            c[row[i]][k] += value[i] * b[col[i]][k];
        }
    }
}
```

第二种算法的 SSE 并行化如下：

```
void coo_multiply_matrix_sse2(int nonzeros, int n, int* row,
int* col, float* value, float** b, float** c) {
    __m128 t1, t2, t3, sum;
    int choice = n % 4;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;

    for (int i = 0; i < nonzeros; i++)
    {
        for (int k = 0; k < n - choice; k += 4)
        {
            t1 = _mm_load_ps(b[col[i]] + k);
            sum = _mm_setzero_ps();
            t3 = _mm_set_ps1(value[i]);
            t2 = _mm_load_ps(c[row[i]] + k);
            sum = _mm_mul_ps(t3, t1);
            t2 = _mm_add_ps(t2, sum);
            _mm_store_ps(c[row[i]] + k, t2);
        }
        for (int k = n - choice; k < n; k++) {
            c[row[i]][k] += value[i] * b[col[i]][k];
        }
    }
```

很明显的，从访存的角度考虑，第二种算法利用 cache 进行优化，将原来不连续的内存转化为连续的内存，在进行并行化的时候，减少了访存的次数，充分利用了向量化的特点，对比第一种算法有明显的优势，在后面的测试数据中我们也能验证。

在这里我们采用第二种算法进行进一步优化，我们实现了 AVX 的 8 路并行，代码如下：

```
void coo_multiply_matrix_avx(int nonzeros, int n, int* row,
int* col, float* value, float** b, float** c) {
    __m256 t1, t2, t3, sum;
    int choice = n % 8;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;

    for (int i = 0; i < nonzeros; i++)
    {
        for (int k = 0; k < n - choice; k += 8)
        {
            sum = _mm256_setzero_ps();
            t1 = _mm256_load_ps(b[col[i]] + k);
            t2 = _mm256_load_ps(c[row[i]] + k);
            t3 = _mm256_set1_ps(value[i]);
            sum = _mm256_mul_ps(t3, t1);
            t2 = _mm256_add_ps(t2, sum);
            _mm256_store_ps(c[row[i]] + k, t2);
        }
        for (int k = n - choice; k < n; k++) {
            c[row[i]][k] += value[i] * b[col[i]][k];
        }
    }

```

但是 avx 八路并行在实验测试当中效果并不理想，在不同的数据规模下都比串行算法要更慢，猜想应该是 avx 在进行向量化的过程中进行的访存次数超过了之前串行算法的次数，导致最后的并行效果不佳。由于笔者笔记本为 AMD 类型的 CPU，可能不支持 512 位的 AVX 指令，笔者多次尝试仍然无法编译通过。

3.2.2 ARM 平台下 neon 并行优化编程

在 ARM 架构下实现并行化思路与 x86 下类似，neon 指令格式也与 sse 和 avx 类似，笔者在这里不做过多介绍，代码如下：

```
void coo_multiply_matrix_neon4(int nonzeros,int n,int* row,
int* col,float* value,float**b,float**c){
    float32x4_t t1,t2,t3,sum;
    int choice = n % 4;
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            c[i][j]=0;

    for (int i=0;i<nonzeros;i++)
    {
        for(int k=0;k<n-choice;k+=4)
        {
            sum=vdupq_n_f32(0.f);
            t1=vld1q_f32(b[col[i]]+k);
            t2=vld1q_f32(c[row[i]]+k);
            t3 = vdupq_n_f32(value[i]);
            sum = vmulq_f32(t3,t1);
            t2= vaddq_f32(t2,sum);
            vst1q_f32(c[row[i]]+k,t2);
        }
        for(int k=n-choice;k < n;k++){
            c[row[i]][k] += value[i] * b[col[i]][k];
        }
    }

```

4 算法性能测试与比较

我们在不同的平台和不同参数下对算法进行性能测试，在矩阵/向量的规模上我们选取了 1024——9192 规模下的数据作为参考，在矩阵的稀疏度的选取上，我们选取了 0.001——0.01 的矩阵稀疏度范围来进行测试，每次测试的时间为 10 次。

对于上面测试的数据，所得的测试结果如下：

对于 SpMV 算法，我们发现无论是 ARM 架构还是 x86 架构，SIMD 并行算法的效率相对于平凡算法都慢很多，而且随着矩阵规模的扩大和矩阵稀疏率的上升，SIMD 并行算法的效率相比于平凡算法都在降低，显然这不是我们期待的并行算法所要达到的效果。

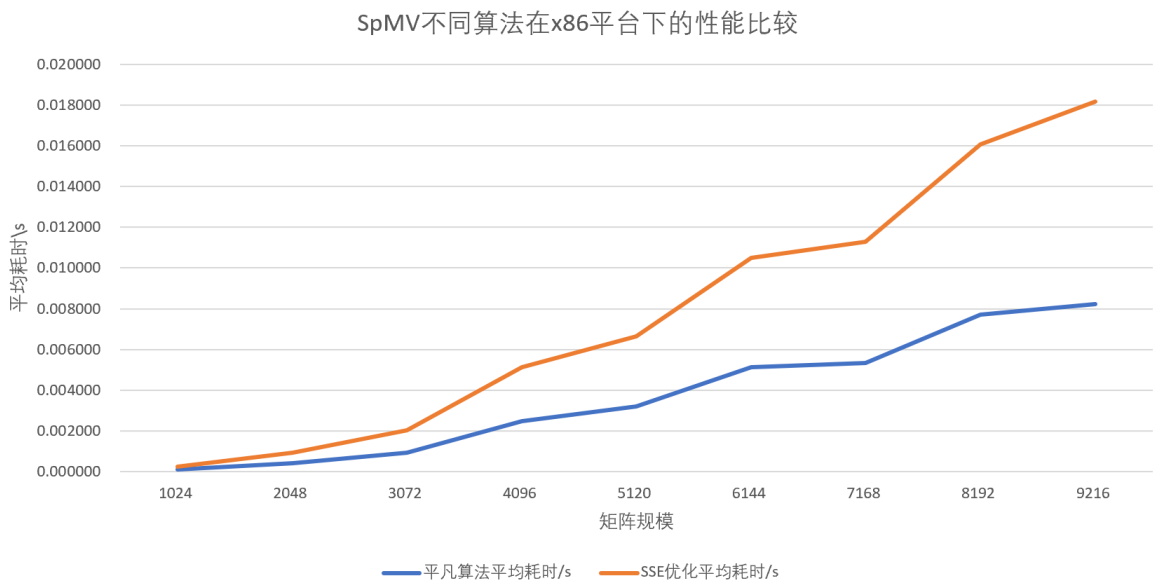

图 4.2: SpMV 不同算法在 x86 平台下的性能比较

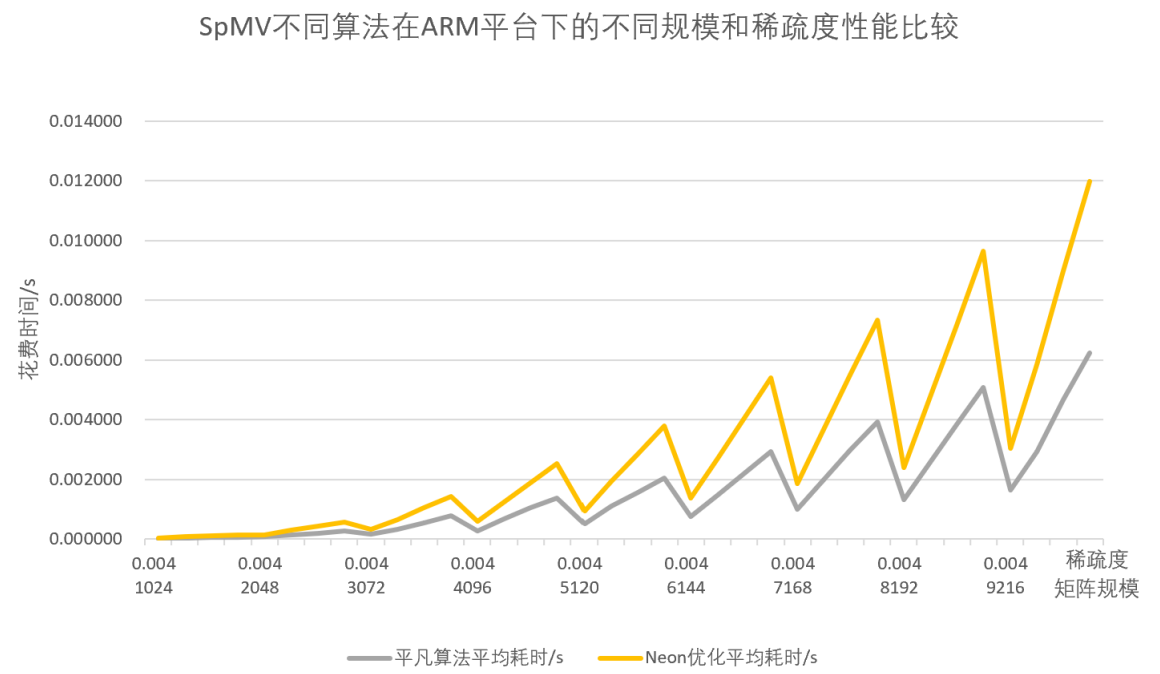

图 4.3: SpMV 不同算法在 ARM 平台下的性能比较

下面我们来观察关于 SpMM 几种算法在不同平台下的性能的差异:

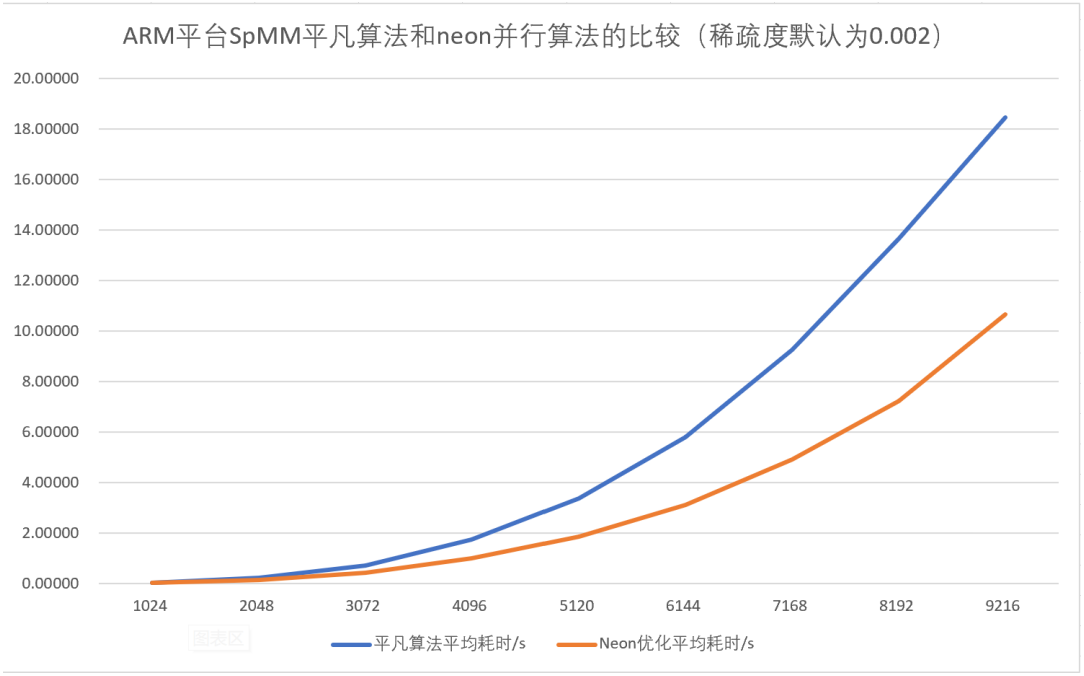

图 4.4: SpMM 不同算法在 ARM 平台上的性能比较

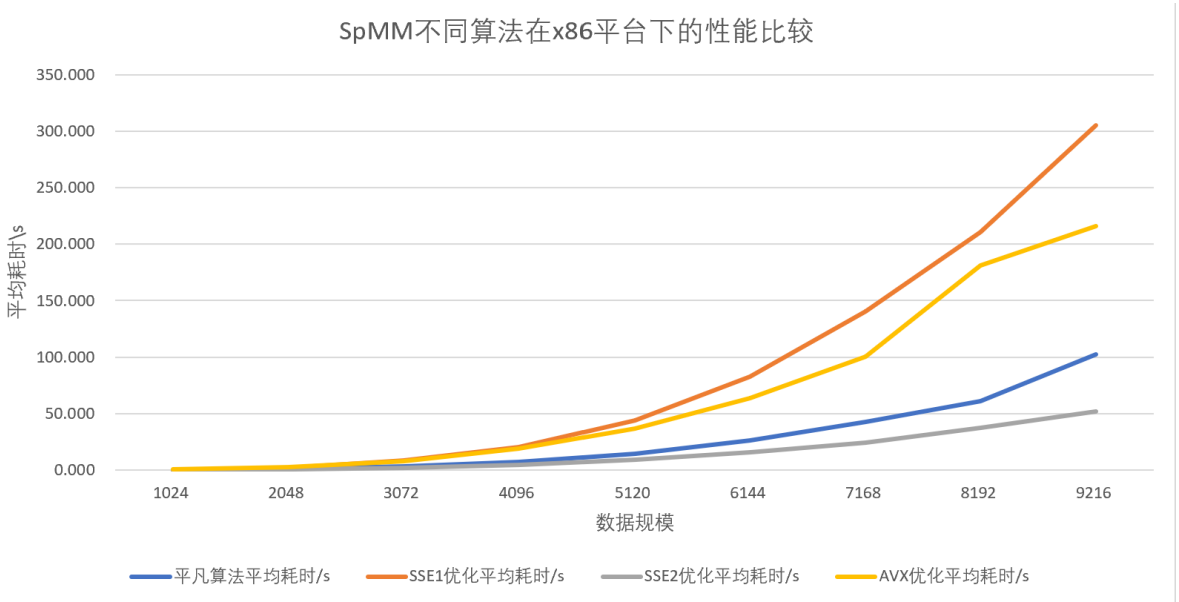

图 4.5: SpMM 不同算法在 x86 平台上的性能比较

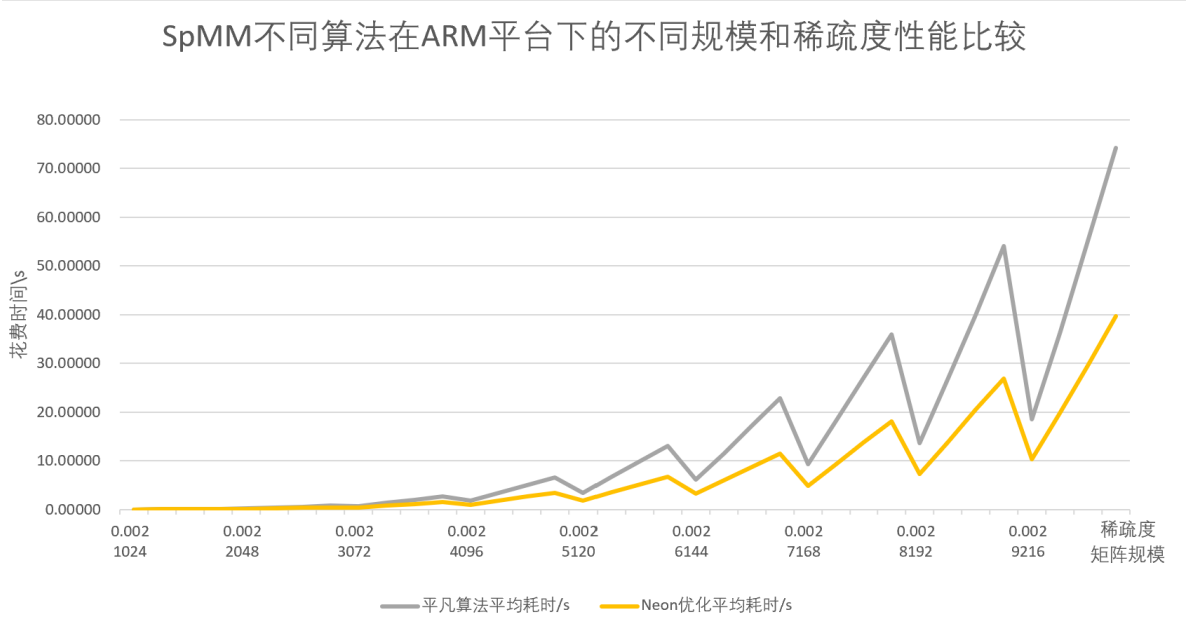

图 4.6: SpMM 不同算法在 ARM 平台上关于矩阵规模和稀疏度的性能比较

与 SpMV 刚好相反，无论是 ARM 架构还是 x86 架构，cache 优化后的 SpMM 的并行算法都要比平凡算法以及未被 cache 优化的并行算法有优势，且随着矩阵规模的扩大，SIMD 并行算法的效率相比于平凡算法提升越来越明显，这是我们期待看到的结果。

将比例进行拟合，可以发现大致呈线性增长的趋势，在测试数据集的规模之下，始终维持着 1.5—2 倍的加速效果，说明我们的并行思路可行，而且取得了可喜的成果。

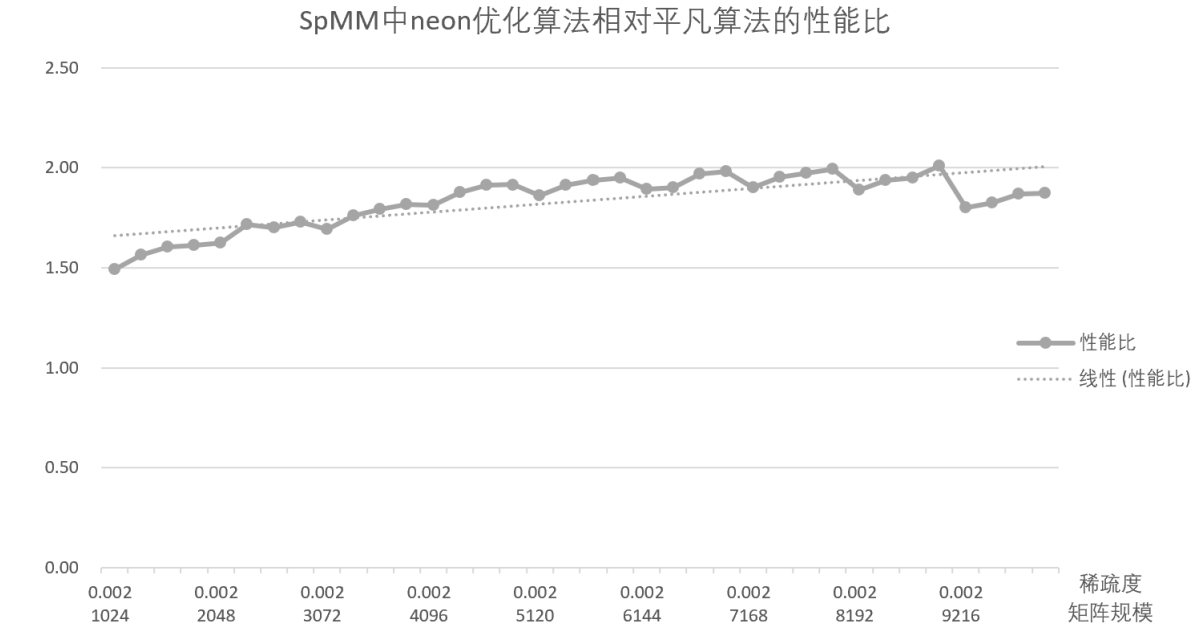

图 4.7: SpMM 在 ARM 平台上并行算法相对平凡算法的加速比

遗憾的是，受笔者水平所限，以及笔记本硬件等条件限制，笔者无法进行更进一步的 vTune 工具的性能测试，使用 perf 命令对于并行化的程序笔者发现无法分析 cache 命中率。

5 代码上传

全部代码素材已上传GitHub.