队伍：iDC-Shizhan的报告文档

1. 总体思路和方法
2. 详细算法设计与实现
   2.1. 矩阵乘法优化
      2.1.1. 循环重排序
      2.1.2. AVX向量化
   2.2. 线程并行优化
   2.3. 存储形式优化
3. 程序代码模块说明
4. 实验结果与分析
5. 代码编译和运行说明

1. 总体思路和方法

GCN优化的思路主要是包括以下几个方面。

- **矩阵乘法上**，优化循环次序、使用AVX指令向量化等
- **并行计算上**，利用OpenMP利用多线程优化计算，同时注意写操作冲突问题。
- **存储形式上**，在someprocessing阶段把存储格式从raw_graph改为CSR格式。

2. 详细算法设计与实现

2.1. 矩阵乘法优化

2.1.1. 循环重排序

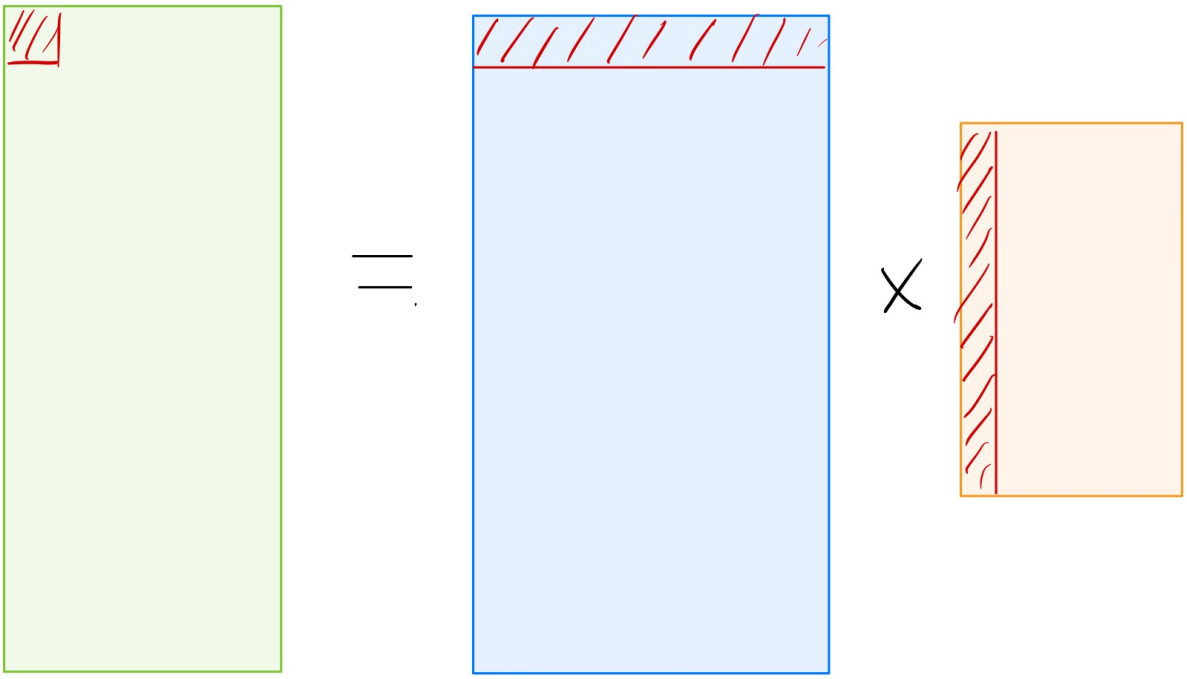

图1 默认循环顺序

如图1所示，使用i,j,k的循环顺序，需要对第二个相乘矩阵（图中橙色矩阵）进行跳跃访问，这是降低效率的。而如果改为i,k,j的循环顺序，那么空间局部性可以得到更好的保证。

循环重排序后 C++ 复制代码

```
for (int i = 0; i < v_num; i++)
    {
        for (int k = 0; k < in_dim; k++)
        {
            for(int j = 0;j < n;j++){
                ……
                ……
            }
        }
    }
```

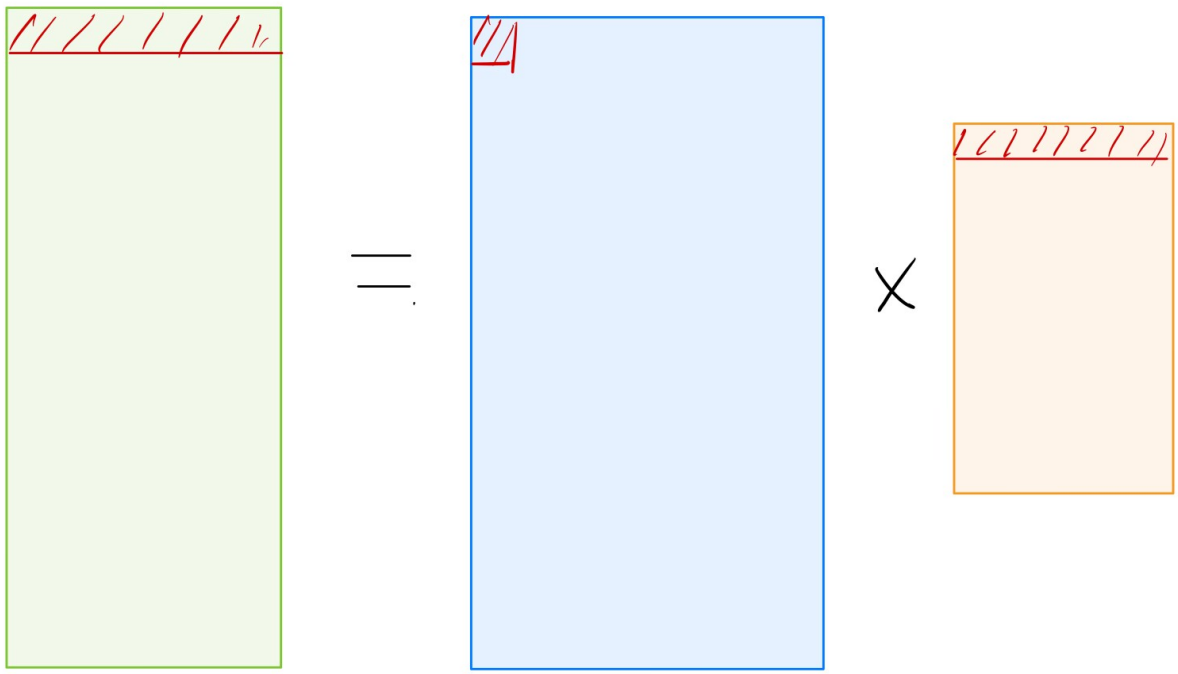

图2 修改循环顺序后

如图2所示，在循环重排序后，对各个矩阵都是连续访问，空间局部性可以得到更好的保证。

2.1.2. AVX向量化

AVX（Advanced Vector Extensions）是一种向量化指令集，用于在现代CPU中执行单指令多数据（SIMD）操作。通过使用AVX指令集，可以将多个数据元素同时处理，从而加速计算。矩阵乘法的过程中包括大量的乘法和加法操作，所以适合使用SIMD进行优化。

向量化示例 C++ 复制代码

```
for(int j=0;j<n;j++){
    __m256 mul_in_w=_mm256_mul_ps(x,_mm256_loadu_ps
    (reinterpret_cast<float const*>(&(tmp_W[k][j*SIMD_NUM]))));
    __m256 old_out=_mm256_loadu_ps
    (reinterpret_cast<float const*>(&(tmp_out_X[i][j*SIMD_NUM])));
    _mm256_storeu_ps(&(tmp_out_X[i][j*SIMD_NUM]),_mm256_add_ps(old_out,mul_
in_w));
}
```

以上是本次程序中使用AVX向量化的一个例子，如图2中，首先取出蓝色矩阵快中的1个元素复制到一个8个元素的向量中，分别与橙色矩阵块的8个连续元素相乘在加到目标绿色矩阵上，这就完成了一次乘法和加法操作，提高了效率。

2.2. 线程并行优化

如果使用单线程编程，那么CPU的资源没有完全利用，可以使用openMP进行多线程编程。首先使用如下语句获取当前系统CPU线程数:

auto max_threads=std::thread::hardware_concurrency();

再使用#pragma omp parallel for语句利用多线程进行编程

#pragma omp parallel for num_threads(max_threads-1)

在多线程并行的过程中，需要注意写操作冲突问题，如以下代码块所示，这是 convertToCSR()函数的部分内容。在转化为CSR格式的过程中，可能会对rowPtr数组多线程进行写操作，会丢失对数组数据的修改。所以我使用了openMP的原子操作atomic，它与临界区critical可以实现相同效果，但原子操作效率会更高一些。

原子操作 C++ 复制代码

```
#pragma omp atomic
        rowPtr[src + 1]++;
        colIndex[i] = dst;
```

2.3. 存储形式优化

在执行计算前将图存储格式改为CSR格式，CSR格式的核心思想是将稀疏矩阵按行进行压缩存储，从而节约存储空间和提高计算效率。

相较于邻接表，CSR具有以下的优势

1. **Cache友好性**：CSR格式具有连续的内存布局，因此在访问节点的邻居信息时，往往会有更好的局部性，减少了内存访问的开销。这样可以更好地利用CPU高速缓存，降低了缓存不命中的概率，从而提高运算效率。
2. **高效的节点邻居访问**：在GNN中，经常需要遍历节点的邻居来执行图卷积等运算。在CSR格式中，对于每个节点，邻居信息已经预先整理为紧凑的形式，因此能够快速高效地获取节点的邻居。
3. **更少的内存占用**：相比邻接表格式，CSR格式通常在存储图结构时需要更少的内存，因为它不需要额外的指针存储。

3. 程序代码模块说明

convertToCSR()　　C++　　复制代码

```cpp
void convertToCSR() {
    rowPtr.resize(v_num + 1, 0);
    colIndex.resize(e_num);
    edge_value.resize(e_num);

    #pragma omp parallel for num_threads(max_threads-1)
    for (int i = 0; i < raw_graph.size() / 2; i++) {
        auto t_id=std::this_thread::get_id();

        int src = raw_graph[2 * i];
        int dst = raw_graph[2 * i + 1];

    #pragma omp atomic
        rowPtr[src + 1]++;
        colIndex[i] = dst;
        // Assign edge value if needed: edge_value[i] = ...;
    }

    for (int i = 1; i <= v_num; i++) {
        rowPtr[i] += rowPtr[i - 1];
    }
}
```

convertToCSR()函数将raw_graph格式的原始图改为CSR格式，主要的数据结构包括rowPtr和colIndex两个数组。

C++ 复制代码

```cpp
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W)
{
    float(*tmp_in_X)[in_dim] = (float(*)[in_dim])in_X;
    float(*tmp_out_X)[out_dim] = (float(*)[out_dim])out_X;
    float(*tmp_W)[out_dim] = (float(*)[out_dim])W;


    const int SIMD_NUM=8;
    int n = out_dim/SIMD_NUM;//循环次数
    #pragma omp parallel for num_threads(max_threads-1)
    for (int i = 0; i < v_num; i++)
    {
        for (int k = 0; k < in_dim; k++)
        {
            //__m256 x=_mm256_set1_ps(tmp_in_X[i][k]);
            __m256 x=_mm256_broadcast_ss(reinterpret_cast<float const*>
            (&(tmp_in_X[i][k])));
            for(int j=0;j<n;j++){
                __m256 mul_in_w=_mm256_mul_ps
                (x,_mm256_loadu_ps(reinterpret_cast<float const*>(&(tmp_W
[k][j*SIMD_NUM]))));
                __m256 old_out=_mm256_loadu_ps(reinterpret_cast<float cons
t*>
                (&(tmp_out_X[i][j*SIMD_NUM])));
                _mm256_storeu_ps(&(tmp_out_X[i][j*SIMD_NUM]),_mm256_add_ps
                    (old_out,mul_in_w));
            }
            //剩下的零散部分处理
            for (int j = n*SIMD_NUM; j < out_dim; j++)
            {
                tmp_out_X[i][j] += tmp_in_X[i][k] * tmp_W[k][j];
            }
        }
    }
}
```

在XW()函数中，增加了多线程并行和SIMD指令的过程，特别的，如果有不能放入整个向量的零散部分则单独计算。

C++ 复制代码

```cpp
void AX(int dim, float* in_X, float* out_X)
{
    float(*tmp_in_X)[dim] = (float(*)[dim])in_X;
    float(*tmp_out_X)[dim] = (float(*)[dim])out_X;

    const int SIMD_NUM=8;
    int n=dim/SIMD_NUM;
    #pragma omp parallel for num_threads(max_threads-1) // schedule(dynamic)
    for (int i = 0; i < v_num; i++)
    {
        int start = rowPtr[i];
        int end = rowPtr[i + 1];
        for (int j = start; j < end; j++) //nbr是i的邻居节点
        {
            int nbr = colIndex[j];
            //__m256 w=_mm256_set1_ps(edge_value[j]);
            __m256 w=_mm256_broadcast_ss
            (reinterpret_cast<float const*>(&(edge_value[j])));
            for (int k=0;k<n;k++){
                __m256 in=_mm256_loadu_ps
                (reinterpret_cast<float const*>(&(tmp_in_X[nbr][k*SIMD_NUM])));
                __m256 out=_mm256_loadu_ps(reinterpret_cast<float const*>
                (&(tmp_out_X[i][k*SIMD_NUM])));
                _mm256_storeu_ps(&(tmp_out_X[i][k*SIMD_NUM]),_mm256_add_ps
                    (_mm256_mul_ps(in,w),out));
            }
            for (int k = SIMD_NUM*n; k < dim; k++)
            {
                tmp_out_X[i][k] += tmp_in_X[nbr][k] * edge_value[j];
            }
        }
    }
}

```

AX()函数和XW相似，都是使用了多线程并行和SIMD指令。

在edgeNormalization()、LogSoftmax()、Relu()等函数中也修改了使用CSR格式、使用openMP并行计算的一些操作，和前面的使用方法类似。

4. 实验结果与分析

实验过程中编写代码构建了v_num=100000，e_num=400000，F0=128，F1=64, F2=16的实验数据。

实验环境是一个16线程的个人电脑，系统版本Ubuntu22.04。

实现结果如下表所示

程序状态	加速比
原程序	1x
增加'修改循环顺序'步骤	1.26x
增加'修改CSR格式'步骤	1.27x
增加'SIMD指令'步骤	3.94x
增加'openMP线程并行'步骤	22.62x

实验结果显示，几种优化手段都可以对GCN计算加速，其中SIMD指令和线程并行操作效果最为明显，最终实现了22.62倍的加速。

5. 代码编译和运行说明

makefile　　Makefile | 复制代码

```
all:
g++ -mfma source_code.cpp -o ../IDC-Shizhan.exe -fopenmp
```

上图为makefile文件，在编译过程中，编译选项-mfma为使用AVX指令所需要的， -fopenmp为使用openMP库进行线程并行所需要的。

最终运行IDC-Shizhan.exe并给出相应的参数即可。