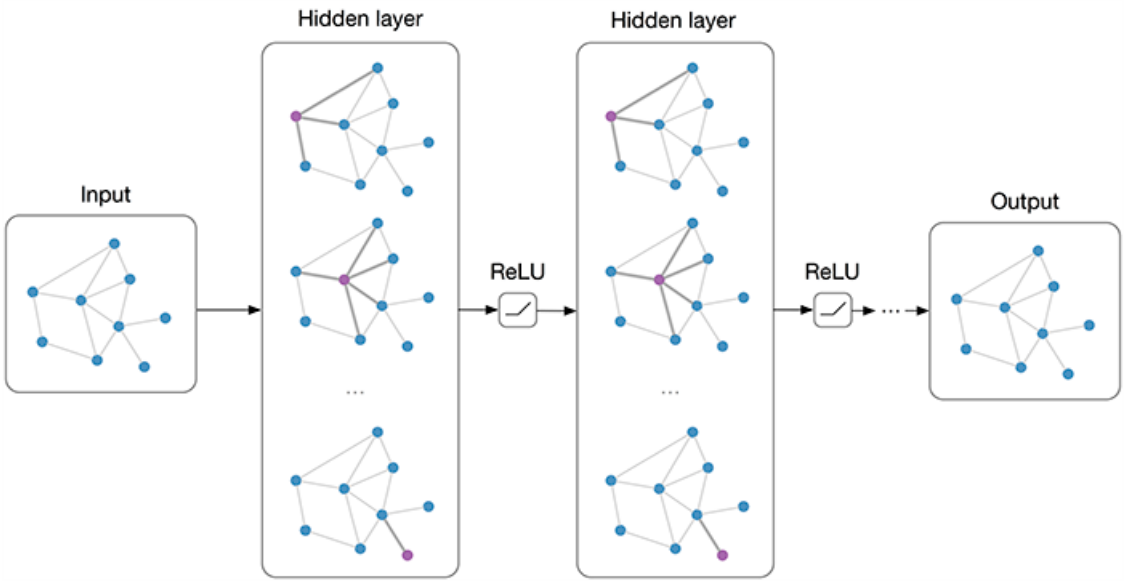
技术报告文档

一、基本算法介绍

本次比赛中使用GCN图卷积神经网络作为基本算法，GCN固定由两个图卷积层构成，第一层的激活函数使用ReLU函数，第二层激活函数使用LogSoftmax函数。



单个图卷积层的传播公式如下：

$$X^{(l+1)} = \alpha\left(\hat{A}X^{(l)}W^{(l)}\right)$$

其中，l表示层号，α为激活函数，$\hat{A} \in \mathbb{R}^{|V|\times|V|}$为归一化后的图邻接矩阵，$X^{(l)} \in \mathbb{R}^{|V|\times F_l}$为输入顶点矩阵特征，$W^{(l)} \in \mathbb{R}^{F_l\times F_{l+1}}$为权重矩阵，$X^{(l+1)} \in \mathbb{R}^{|V|\times F_{l+1}}$为输出顶点特征矩阵。归一化邻接矩阵的计算公式为$\hat{A} = D^{-1/2}AD^{-1/2}$，其中A为图邻接矩阵（默认包含自环），D为A对应的顶点度矩阵（对角矩阵）。

ReLU函数定义为：

$$ReLU(x) = max(0, x)$$

函数实现为：

```
void ReLU(int dim, float *X)
{
    for (int i = 0; i < v_num * dim; i++)
        if (X[i] < 0)
            X[i] = 0;
}
```

LogSoftmax函数定义为：

$$\mathrm{LogSoftmax}\left(\mathrm{X}_{\mathrm{i,j}}^{(\mathrm{l})}\right) = \left(\mathrm{X}_{\mathrm{i,j}}^{(\mathrm{l})} - \mathrm{X}_{\mathrm{i,max}}^{(\mathrm{l})}\right) - \log\left(\sum_{c=0}^{F_l-1} e^{\mathrm{X}_{\mathrm{i,c}}^{(\mathrm{l})} - \mathrm{X}_{\mathrm{i,max}}^{(\mathrm{l})}}\right)$$

$$\mathrm{X}_{\mathrm{i,max}}^{(\mathrm{l})} = \max\left(\mathrm{X}_{\mathrm{i,0}}^{(\mathrm{l})}, \ldots, \mathrm{X}_{\mathrm{i},F_l-1}^{(\mathrm{l})}\right)$$

函数实现为：

```
void LogSoftmax(int dim, float *X)
{
    for (int i = 0; i < v_num; i++)
    {
        float max = X[i * dim];
        for (int j = 1; j < dim; j++)
        {
            if (X[i * dim + j] > max)
                max = X[i * dim + j];
        }

        float sum = 0;
        for (int j = 0; j < dim; j++)
        {
            sum += std::exp(X[i * dim + j] - max);
        }
        sum = std::log(sum);

        for (int j = 0; j < dim; j++)
        {
            X[i * dim + j] = X[i * dim + j] - max - sum;
```

```
        }
    }
}
```

二、设计思路和方法

本次比赛的目标是使用优化算法进行GCN图推理计算的加速优化，考虑到矩阵乘法计算的并行性，我们使用OpenMP多线程优化算法和AVX指令集优化来加速图推理任务，通过将任务分解为多个并行执行的子任务，充分利用多核CPU的计算能力。

首先是OpenMP多线程优化算法，我们使用OpenMP的'#pragma omp parallel for'指令将任务拆分为多个并行的循环任务，并且确保任务之间没有数据依赖性，以免出现数据竞争和死锁。我们通过设置变量num_threads来控制并行区域中的线程数量，以适应不同的硬件配置。对于可能引起数据竞争的临界区，我们考虑使用'#pragma omp critical'指令来保护，确保同时只有一个线程进入临界区执行。根据任务的复杂性和负载平衡，我们考虑使用OpenMP的'schedule'选项来优化任务的调度方式，如静态调度或动态调度。

其次，我们的目标是利用AVX指令集来加速图推理任务，通过向量化操作同时处理多个数据元素。我们使用AVX指令集提供的SIMD指令来执行向量化操作，就多个数据元素打包到256位的YMM寄存器中，并同时进行计算。我们将任务中的循环操作转换为AVX向量指令，以实现对多个数据元素的并行计算。我们考虑利用AVX指令集中的数据复用功能，将YMM寄存器中的一个数据元素复制到其他位置，减少数据加载和存储的次数，从而减少访存开销。在编程中，我们考虑使用经过优化的AVX库函数，如AVX优化的数学库函数，以提高特定操作的计算效率。在某些情况下，特别是对于浮点数计算，AVX指令集可能引入微小的摄入误差，我们需要考虑确保使用AVX指令集时不会损失计算精度。

对于两种优化算法，我们需要设计一系列实验来评估其计算精度、性能和加速比。通过在相同的硬件平台上执行图推理任务，并记录执行时间，我们可以得出以下结果：

1、分别执行未优化算法、OpenMP多线程优化算法、AVX指令集优化算法、同时进行OpenMP多线程优化和AVX指令集优化算法，并记录它们的计算精度和执行时间。

2、计算并比较不同算法的加速比，即优化算法相对于未优化算法的性能提升倍数。

在进行性能评估时，我们要注意考虑数据规模、并行线程数、硬件平台等因素，以获得更准确的结果。

最后，根据实验结果，我们考虑选择性能最优的优化算法，并进行必要的调优和改进，以达到最佳的图推理性能。

三、算法优化

本次比赛采用两种GCN图推理优化算法，分别是OpenMP多线程优化算法和AVX指令集优化算法。

1、OpenMP多线程优化算法

OpenMP是一种支持多线程并行编程的开放式并行编程标准，允许在C++中通过添加简单的编译器指令来实现并行化。OpenMP的基本原理是通过将代码分成多个任务，然后将这些任务分配给不同的线程来实现并行计算。

在OpenMP中，使用**'#pragma omp parallel for'**指令来定义并行区域。当进入并行区域时，系统会根据硬件的可用处理器核心数创建多个线程，每个线程将执行并行区域中的代码块。在并行区域中，可以使用不同的OpenMP指令来实现并行化。例如，'#pragma omp for'指令用于循环并行化，'#pragma omp sections'指令用于区段并行化，'#pragma omp task'指令用于任务并行化等。

OpenMP采用共享内存的模型，即所有线程共享程序的全局内存。这意味着线程可以互相访问共享的变量，而不需要显式地进行数据传输。

由于多个线程并行执行，可能会出现数据竞争和死锁等问题。OpenMP提供了一些线程同步的机制，如'#pragma omp barrier'指令用于等待所有线程到达同步点，**'#pragma omp critical'**指令用于保护临界区，'#pragma omp atomic'指令用于进行原子操作等。

OpenMP还支持一些环境变量，允许对并行计算进行更精细的控制。例如，可以使用'OMP_NUM_THREADS'环境变量**设置并行区域中的线程数**。

2、AVX指令集优化算法

AVX（Advanced Vector Extensions）指令集是一组由英特尔和AMD共同推出的**SIMD（Single Instruction，Multiple Data）**指令集扩展。AVX指令集可以用于加速同一指令对多个数据元素进行并行处理，从而提高计算性能。AVX指令集主要用于执行单精度浮点数和双精度浮点数的向量运算，以及整数向量运算。

使用AVX指令集可以执行向量化操作，AVX指令集引入了256位的YMM寄存器，**可以同时处理8个单精度浮点数**或4个双精度浮点数。通过将多个数据元素打包到一个寄存器中，可以实现对这些数据的享量化操作，即同时执行相同的指令对多个数据元素进行计算，从而大大提升计算效率。

AVX指令集中的SIMD指令可以同时对多个数据元素执行相同的计算操作，而不是逐个进行处理。这样，即使在单个指令周期内，也可以并行计算多个数据，从而加速整体计算过程。

AVX指令集允许将YMM寄存器中的一个数据元素复制到其他位置，实现数据复用，从而减少数据加载和存储的次数，减少访存开销，进一步提高计算性能。

AVX指令集中的浮点运算指令支持更多的操作类型和运算模式，如乘加指令、平方根指令等，这些指令能够高效地执行复杂的浮点运算，从而加速计算过程。

使用AVX指令集可以在不损失计算精度的情况下加速计算过程，尤其适用于涉及大量数据运算的科学计算和图形处理等应用。

四、详细算法设计与实现

1、OpenMP多线程优化算法

在全局变量中定义并行执行的线程数。

```
int num_threads = 0;              //并行执行的线程数
```

在使用OpenMP多线程优化之前，先查看CPU内核数。

```
//显示处理器核心数，设置线程数为32
int numProcs = omp_get_num_procs();
//std::cout << "Number of available CPU cores: " << numProcs << std::endl;
num_threads = 32;
```

在函数XW和函数AX中，设置并行执行的线程数。

```
omp_set_num_threads(num_threads);
```

函数XW采用OpenMP多线程方法实现。

```
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W)
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
        for (int j = 0; j < out_dim; j++)
        {
            for (int k = 0; k < in_dim; k++)
            {
                out_X[i * out_dim + j] += in_X[i * in_dim + k] * W[k * out_dim + j];
            }
        }
    }
}
```

函数AX采用OpenMP多线程方法实现。

```
void AX(int dim, float *in_X, float *out_X)
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
        std::vector<int> &nlist = edge_index[i];
        for (int j = 0; j < nlist.size(); j++)
        {
```

```
            int nbr = nlist[j];
            for (int k = 0; k < dim; k++)
            {
#pragma omp atomic
                out_X[i * dim + k] += in_X[nbr * dim + k] * edge_val[i][j];
            }
        }
    }
}
```

2、AVX指令集优化算法

引入包含AVX指令集的头文件。

```
#include <immintrin.h>
```

设置GCC编译器将代码优化为使用AVX2指令集。AVX2可以对256位的YMM寄存器进行并行处理，从而加快向量运算。

设置GCC编译器对代码进行03级别的优化和循环展开优化。03级别是GCC编译器中的最高级别优化，会对代码进行多种优化技术，包括内联函数、代码移动、数据流分析等，以提高代码性。循环展开优化可以将循环中的迭代次数展开成多个重复的代码块，从而减少循环控制开销，提高循环体内部的计算效率。

```
pragma GCC target("avx2")
#pragma GCC optimize("O3","unroll-loops")
```

函数XW采用AVX指令集方法实现。

```
/*AVX指令集*/
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W)
{
    // 每次处理8个单精度浮点数
    for (int i = 0; i < v_num; i++)
    {
        for (int j = 0; j < out_dim; j += 8)
        {
            __m256 sum = _mm256_setzero_ps(); // 初始化累加和为零向量

            for (int k = 0; k < in_dim; k++)
            {
                // 加载输入矩阵和权重矩阵的向量
                __m256 X_vec = _mm256_loadu_ps(&in_X[i * in_dim + k]);
                __m256 W_vec = _mm256_loadu_ps(&w[k * out_dim + j]);

                // 执行乘法运算
                __m256 mul = _mm256_mul_ps(X_vec, W_vec);

                // 累加乘积结果
                sum = _mm256_add_ps(sum, mul);
```

```
            }

            // 存储累加和到输出矩阵
            _mm256_storeu_ps(&out_X[i * out_dim + j], sum);
        }
    }
}
```

函数AX采用AVX指令集方法实现。

```
/*AVX指令集*/
void AX(int dim, float *in_X, float *out_X)
{
    for (int i = 0; i < v_num; i++)
    {
        std::vector<int> &nlist = edge_index[i];
        for (int j = 0; j < nlist.size(); j += 8) // 假设使用AVX-256，一次处理8个元素
        {
            int nbr[8];
            for (int k = 0; k < 8; k++)
            {
                if (j + k < nlist.size())
                    nbr[k] = nlist[j + k];
                else
                    nbr[k] = -1; // 用于处理边界情况，如果邻居索引不足8个，则填充-1
            }

            for (int k = 0; k < dim; k += 8) // 一次处理8个维度
            {
                __m256 in_vec = _mm256_loadu_ps(&in_X[i * dim + k]); // 加载输入向量中
的数据
                __m256 out_vec = _mm256_loadu_ps(&out_X[i * dim + k]); // 加载输出向量
中的数据

                for (int l = 0; l < 8; l++)
                {
                    if (nbr[l] != -1)
                    {
                        __m256 edge_val_vec = _mm256_set1_ps(edge_val[i][j + l]); //
使用边界矩阵中的值创建一个向量
                        __m256 in_X_nbr_vec = _mm256_loadu_ps(&in_X[nbr[l] * dim +
k]); // 加载邻居节点的输入向量数据

                        __m256 result_vec = _mm256_mul_ps(in_X_nbr_vec,
edge_val_vec); // 对应元素相乘
                        out_vec = _mm256_add_ps(out_vec, result_vec); // 累加到输出向
量中
                    }
                }
```

```
                _mm256_storeu_ps(&out_X[i * dim + k], out_vec); // 存储更新后的输出向量
数据
            }
        }
    }
}
```

3、同时使用OpenMP多线程和AVX指令集进行优化

函数XW同时使用OpenMP多线程和AVX指令集进行优化实现。

```
/*多线程+AVX指令集*/
void XW(int in_dim, int out_dim, float* in_X, float* out_X, float* W)
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
        for (int j = 0; j < out_dim; j += 8)
        {
            __m256 sum = _mm256_setzero_ps(); // 初始化累加和为零向量

            for (int k = 0; k < in_dim; k++)
            {
                __m256 X_vec = _mm256_loadu_ps(&in_X[i * in_dim + k]);
                __m256 W_vec = _mm256_loadu_ps(&w[k * out_dim + j]);

                __m256 mul = _mm256_mul_ps(X_vec, W_vec);

                sum = _mm256_add_ps(sum, mul);
            }

            // 存储累加和到输出矩阵
            _mm256_storeu_ps(&out_X[i * out_dim + j], sum);
        }
    }
}
```

函数AX同时使用OpenMP多线程和AVX指令集进行优化实现。

```
/*多线程+AVX指令集*/
void AX(int dim, float* in_X, float* out_X)
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
```

```
        std::vector<int>& nlist = edge_index[i];
        for (int j = 0; j < nlist.size(); j += 8)
        {
            int nbr[8];
            for (int k = 0; k < 8; k++)
            {
                if (j + k < nlist.size())
                    nbr[k] = nlist[j + k];
                else
                    nbr[k] = -1;
            }

            for (int k = 0; k < dim; k += 8)
            {
                __m256 in_vec = _mm256_loadu_ps(&in_X[i * dim + k]);
                __m256 out_vec = _mm256_loadu_ps(&out_X[i * dim + k]);

                for (int l = 0; l < 8; l++)
                {
                    if (nbr[l] != -1)
                    {
                        __m256 edge_val_vec = _mm256_set1_ps(edge_val[i][j + l]);
                        __m256 in_X_nbr_vec = _mm256_loadu_ps(&in_X[nbr[l] * dim +
k]);

                        __m256 result_vec = _mm256_mul_ps(in_X_nbr_vec,
edge_val_vec);
                        out_vec = _mm256_add_ps(out_vec, result_vec);
                    }
                }

                _mm256_storeu_ps(&out_X[i * dim + k], out_vec);
            }
        }
    }
}
```

五、实验结果与分析

用于测试的图数据规模如下（其中数字表示作为测试集的相应规模图的个数）。

顶点\边	<500K	<1M	<5M
<500K	1	1	2
<1M		1	1
<5M			1

GCN模型：

$$F_0 \leq 128, F_1 = 16, F_2 \leq 32.$$

测试实验环境：

- CPU：Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz（物理核心数160，逻辑核心数320）
- 内存：251GB
- 操作系统：Linux
- g++或gcc编译器：8.3.0

测试实验使用Pytorch自行生成随机图文件、随机顶点特征及随机权重矩阵进行测试，测试结果如下。

Table 1: max_sum

graph_size	example	OpenMP(num_thd=32)	AVX	OpenMP&AVX
1024*2048	-16.68968964	-16.68968964	-16.65371513	-16.65371513
500K*500K	-16.63553047	-16.63553047	-16.63553047	-16.63553047
500K*1M	-16.63553047	-16.63553047	-16.63552856	-16.63552856
500K*5M1	-16.63553047	-16.63553047	-16.63553047	-16.63553047
500K*5M2	-16.63553047	-16.63553047	-16.63553047	-16.63553047
1M*1M	-16.63553047	-16.63553047	-16.63553047	-16.63553047
1M*5M	-16.63553047	-16.63553047	-16.63553047	-16.63553047
5M*5M	-16.63553047	-16.63553047	-16.63552856	-16.63552856

Table 2: l_timeMs(SpeedUp)

graph_size	example	OpenMP(num_thd=32)	AVX	OpenMP&AVX
1024*2048	18.76	11.09 (×**1.69**)	8.00 (×**2.34**)	8.45 (×**2.22**)
500K*500K	1973.79	599.91 (×**3.29**)	538.49 (×**3.67**)	433.47 (×**4.55**)
500K*1M	4044.60	1233.15 (×**3.28**)	1149.18 (×**3.52**)	904.54 (×**4.47**)
500K*5M1	5091.33	2086.72 (×**2.44**)	1922.89 (×**2.65**)	1519.05 (×**3.35**)
500K*5M2	7651.71	4246.07 (×**1.80**)	3329.77 (×**2.30**)	2826.08 (×**2.71**)
1M*1M	7050.62	1477.50 (×**4.77**)	1562.73 (×**4.51**)	1142.30 (×**6.17**)
1M*5M	11790.23	5284.76 (×**2.23**)	4919.57 (×**2.40**)	3860.63 (×**3.05**)
5M*5M	35353.030	7836.47 (×**4.51**)	8672.38 (×**4.08**)	5925.59 (×**5.97**)

其中，max_sum为最大的顶点特征矩阵行和，l_timeMs为执行时间（采用程序多次运行的平均结果，单位为ms），SpeedUp为加速比。

实验结果显示，OpenMP多线程优化算法在边分布较为稀疏的图上加速效果最好，可达到**4.5倍左右的加速比**；OpenMP多线程优化算法在边分布较为稠密的图上加速效果可达到**2倍左右的加速比。**

使用AVX指令集优化算法，引入256位寄存器，可以同时处理更多的数据，从而得到**至少两倍的加速比**，在边分布较为稀疏的图上可达到**4.5倍左右的加速比。**

当同时采用OpenMP多线程优化算法和AVX指令集进行优化时，可以得到**2.7倍~6.2倍的加速比**，性能优化较为明显。

六、程序代码模块说明

查看处理器核心数，设置多线程数。

```
int numProcs = omp_get_num_procs();
//std::cout << "Number of available CPU cores: " << numProcs << std::endl;
num_threads = 32;
```

输入七个参数，不计算读文件（reading files）、分配内存（malloc）和初始化内存（memset）的时间。

```
F0 = atoi(argv[1]);                    //输入层特征长度，其中atoi()函数用于将字符串转换为整数类型。
F1 = atoi(argv[2]);                    //第一层特征长度
F2 = atoi(argv[3]);                    //第二层特征长度
readGraph(argv[4]);                    //图结构（文件名）
readFloat(argv[5], X0, v_num * F0);    //X0（输入顶点特征矩阵文件名），矩阵大小为“顶点数×F0”
readFloat(argv[6], W1, F0 * F1);       //W1（第一层权重矩阵文件名），矩阵大小为“F0×F1”
readFloat(argv[7], W2, F1 * F2);       //W2（第二层权重矩阵文件名），矩阵大小为“F1×F2”

initFloat(X1, v_num * F1);
initFloat(X1_inter, v_num * F1);
initFloat(X2, v_num * F2);
initFloat(X2_inter, v_num * F2);
```

主程序，计算程序运行的时间，输出两个值，分别为最大的顶点特征矩阵行和max_sum与执行时间l_timeMs。

```
//计算开始时的时间点
TimePoint start = chrono::steady_clock::now();

//预处理的时间应包括在内
somePreprocessing();
edgeNormalization();

// printf("Layer1 XW\n");
XW(F0, F1, X0, X1_inter, W1);

// printf("Layer1 AX\n");
AX(F1, X1_inter, X1);

// printf("Layer1 ReLU\n");
ReLU(F1, X1);

// printf("Layer2 XW\n");
XW(F1, F2, X1, X2_inter, W2);

// printf("Layer2 AX\n");
AX(F2, X2_inter, X2);

// printf("Layer2 LogSoftmax\n");
```

```
LogSoftmax(F2, X2);

// You need to compute the max row sum for result verification
float max_sum = MaxRowSum(X2, F2);

//计算结束时的时间点
TimePoint end = chrono::steady_clock::now();
chrono::duration<double> l_durationSec = end - start;
double l_timeMs = l_durationSec.count() * 1e3;

// Finally, the max row sum and the computing time
// should be print to the terminal in the following format
printf("%.8f\n", max_sum);
printf("%.8lf\n", l_timeMs);
fflush(stdout);

// Remember to free your allocated memory
freeFloats();
```

导入头文件和定义全局变量。

```
#include <stdio.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <cmath>
#include <string.h>
#include <omp.h>
#include <iostream>
#include <iomanip>
#include <chrono>
#include <omp.h>
#include <thread>
#include <cstdio>
#include <cstdlib>
#include <immintrin.h>           // 包含AVX指令集的头文件
#pragma GCC target("avx2")
#pragma GCC optimize("O3","unroll-loops")

using namespace std;

typedef std::chrono::time_point<std::chrono::steady_clock> TimePoint;

int v_num = 0;                   //顶点数
int e_num = 0;                   //边数
int F0 = 0, F1 = 0, F2 = 0;      //F0 <= 128; F1 = 16; F2 <= 32
int num_threads = 0;             //并行执行的线程数

vector<vector<int>> edge_index;
vector<vector<float>> edge_val;
vector<int> degree;
```

```
vector<int> raw_graph;

float *X0, *W1, *W2, *X1, *X1_inter, *X2, *X2_inter;
```

读取文件必须使用示例文件提供的readGraph()函数，不可修改，不计入执行时间内，若需转换为邻接表或是CSR等格式须在somePreprocessing()函数内实现，并计入执行时间。

```
void readGraph(char *fname)
{
    ifstream infile(fname);

    int source;
    int end;

    infile >> v_num >> e_num;

    raw_graph.resize(e_num * 2);

    while (!infile.eof())
    {
        infile >> source >> end;
        if (infile.peek() == EOF)
            break;
        raw_graph.push_back(source);
        raw_graph.push_back(end);
    }
}
```

预处理部分：转换图向量为邻接矩阵，边的归一化处理。

```
//将原图向量转为邻接矩阵
void raw_graph_to_AdjacencyList()
{
    int src;
    int dst;

    edge_index.resize(v_num);
    edge_val.resize(v_num);
    degree.resize(v_num, 0);

    for (int i = 0; i < raw_graph.size() / 2; i++)
    {
        src = raw_graph[2 * i];
        dst = raw_graph[2 * i + 1];
        edge_index[dst].push_back(src);
        degree[src]++;
    }
}

void somePreprocessing()
{
```

```
    //The graph  will be transformed into adjacency list ,you can use other data
structure such as CSR
    raw_graph_to_AdjacencyList();
}

//边的归一化
void edgeNormalization()
{
    for (int i = 0; i < v_num; i++)
    {
        for (int j = 0; j < edge_index[i].size(); j++)
        {
            float val = 1 / sqrt(degree[i]) / sqrt(degree[edge_index[i][j]]);
            edge_val[i].push_back(val);
        }
    }
}
```

读顶点特征矩阵文件和权重矩阵文件，并进行初始化。

```
void readFloat(const char* fname, float*& dst, int num)
{
    dst = (float*)malloc(num * sizeof(float));

    FILE* fp = fopen(fname, "rb");
    if (fp == nullptr)
    {
        // 文件打开失败，进行错误处理
        printf("无法打开文件：%s\n", fname);
        // 其他错误处理代码
        // ...
    }
    else
    {
        // 文件成功打开，可以进行文件读取操作
        // 读取文件内容
        fread(dst, sizeof(float), num, fp);
        fclose(fp);
    }
}

void initFloat(float *&dst, int num)
{
    dst = (float *)malloc(num * sizeof(float));
    memset(dst, 0, num * sizeof(float));
}
```

对函数XW和函数AX同时使用OpenMP多线程和AVX指令集进行优化实现。

```
/*多线程+AVX指令集*/
void XW(int in_dim, int out_dim, float* in_X, float* out_X, float* W)
```

```
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
        for (int j = 0; j < out_dim; j += 8)
        {
            __m256 sum = _mm256_setzero_ps(); // 初始化累加和为零向量

            for (int k = 0; k < in_dim; k++)
            {
                __m256 X_vec = _mm256_loadu_ps(&in_X[i * in_dim + k]);
                __m256 W_vec = _mm256_loadu_ps(&w[k * out_dim + j]);

                __m256 mul = _mm256_mul_ps(X_vec, W_vec);

                sum = _mm256_add_ps(sum, mul);
            }

            // 存储累加和到输出矩阵
            _mm256_storeu_ps(&out_X[i * out_dim + j], sum);
        }
    }
}

/*多线程+AVX指令集*/
void AX(int dim, float* in_X, float* out_X)
{
    omp_set_num_threads(num_threads);

#pragma omp parallel for
    for (int i = 0; i < v_num; i++)
    {
        std::vector<int>& nlist = edge_index[i];
        for (int j = 0; j < nlist.size(); j += 8)
        {
            int nbr[8];
            for (int k = 0; k < 8; k++)
            {
                if (j + k < nlist.size())
                    nbr[k] = nlist[j + k];
                else
                    nbr[k] = -1;
            }

            for (int k = 0; k < dim; k += 8)
            {
                __m256 in_vec = _mm256_loadu_ps(&in_X[i * dim + k]);
                __m256 out_vec = _mm256_loadu_ps(&out_X[i * dim + k]);

                for (int l = 0; l < 8; l++)
```

```
                {
                    if (nbr[l] != -1)
                    {
                        __m256 edge_val_vec = _mm256_set1_ps(edge_val[i][j + l]);
                        __m256 in_X_nbr_vec = _mm256_loadu_ps(&in_X[nbr[l] * dim +
k]);

                        __m256 result_vec = _mm256_mul_ps(in_X_nbr_vec,
edge_val_vec);
                        out_vec = _mm256_add_ps(out_vec, result_vec);
                    }
                }

                _mm256_storeu_ps(&out_X[i * dim + k], out_vec);
            }
        }
    }
}
```

激活函数ReLU和LogSoftmax的定义。

```
void ReLU(int dim, float *X)
{
    for (int i = 0; i < v_num * dim; i++)
        if (X[i] < 0)
            X[i] = 0;
}

void LogSoftmax(int dim, float *X)
{
    for (int i = 0; i < v_num; i++)
    {
        float max = X[i * dim];
        for (int j = 1; j < dim; j++)
        {
            if (X[i * dim + j] > max)
                max = X[i * dim + j];
        }

        float sum = 0;
        for (int j = 0; j < dim; j++)
        {
            sum += std::exp(X[i * dim + j] - max);
        }
        sum = std::log(sum);

        for (int j = 0; j < dim; j++)
        {
            X[i * dim + j] = X[i * dim + j] - max - sum;
        }
    }
```

```
}
```

计算最大的顶点特征矩阵行和。

```
float MaxRowSum(float *X, int dim)
{
    float max = -__FLT_MAX__;
    for (int i = 0; i < v_num; i++)
    {
        float sum = 0;
        for (int j = 0; j < dim; j++)
        {
            sum += X[i * dim + j];
        }
        if (sum > max)
            max = sum;
    }
    return max;
}
```

释放所有分配的内存。

```
void freeFloats()
{
    free(X0);
    free(W1);
    free(W2);
    free(X1);
    free(X2);
    free(X1_inter);
    free(X2_inter);
}
```

七、详细程序代码编译说明

makefile文件用于编译源代码并生成可执行文件。

```
CC = g++
CFLAGS = -Wall -std=c++11 -march=native -fopenmp
TARGET = ../goodgoodstudy.exe
SRC = source_code.cpp

all: $(TARGET)

$(TARGET): $(SRC)
    $(CC) $(CFLAGS) -o $@ $^

clean:
    rm -f $(TARGET)
```

'CC = g++'：定义编译器为g++编译器。

'CFLAGS = -Wall -std=c++11 -march=native -fopenmp'：定义编译选项。其中，'-Wall'：打开所有警告信息；'-std=c++11'：使用C++11标准进行编译；'-march=native'：根据当前主机的处理器架构自动优化生成的机器码；'-fopenmp'：启用OpenMP多线程支持，用于并行编程。

'TARGET = ../goodgoodstudy.exe'：定义生成的可执行文件名为goodgoodstudy.exe，存储在上一级目录中。

'SRC = source_code.cpp'：定义源代码文件名为source_code.cpp。

下面是makefile规则：

'all'：默认规则，即运行'make'命令时会执行的规则。它依赖于'$(TARGET)'规则，会编译生成可执行文件。

'clean'：清理规则，用于删除生成的可执行文件。执行'make clean'命令会删除目标文件。

八、详细代码运行使用说明

1、GitHub项目文件结构

```
cgc_goodgoodstudy
├── goodgoodstudy
|       ├── source_code.cpp
|       ├── makefile
|       └── README
├── goodgoodstudy_report
|       └── report.pdf
└── goodgoodstudy.exe
```

2、测试方式

打开命令行，进入目录cgc_goodgoodstudy/goodgoodstudy。

```
cd cgc_goodgoodstudy/goodgoodstudy
```

编译makefile文件，在上级目录形成可执行文件。

```
make
```

返回上级目录。

```
cd ..
```

执行可执行文件，传入7个参数，分别是：输入顶点特征长度、第一层顶点特征长度、第二层顶点特征长度、图结构文件名、输入顶点特征矩阵文件名、第一层权重矩阵文件名、第二层权重矩阵文件名。

```
./goodgoodstudy.exe 64 16 8 graph/500K500K.txt embedding/500K500K.bin
weight/W_64_16.bin weight/W_16_8.bin
```

可执行程序打印输出两个值，分别为最大的顶点特征矩阵行和执行时间。

```
[wangyj@node189 cgc_goodgoodstudy]$ ./goodgoodstudy.exe 64 16 8 graph/5M5M.txt embedding/5M5M.bin weight/W_64_16.bin weight/W_16_8.bin
-16.63552856
5887.33824000
```

九、附录

原始实验数据。

1024*2048（1024_example_graph）	max_sum	l_timeMs
example	-16.68968964	18.11891500 19.47389100 16.66450600 19.24614300 20.31048800
多线程（线程数=32）	-16.68968964	11.53425900 11.26136000 10.68568200 11.14905200 10.83201400

1024*2048（1024_example_graph）	max_sum	l_timeMs
AVX指令集	-16.65371513	8.83959200 8.02289600 6.56665300 8.00213100 8.58387100
多线程+AVX指令集	-16.65371513	8.25446500 9.29791800 7.97402100 7.82732700 8.90437800

V250K*E499K（500K500K）	max_sum	l_timeMs
example	-16.63553047	1973.64904600 1973.85338200 1973.26150600 1974.68918900 1973.48291200
多线程（线程数=32）	-16.63553047	609.28922200 596.22916600 596.04655800 594.59570300 603.37234100
AVX指令集	-16.63553047	539.79046200 538.41308400 536.87194600 538.70569200 538.67994500
多线程+AVX指令集	-16.63553047	433.16059300 433.72254600 431.97189500 433.94963500 434.56750200

V499K*E1023K（500K1M）	max_sum	l_timeMs

V499K*E1023K（500K1M）	max_sum	l_timeMs
example	-16.63553047	4050.61365100 4050.97179600 4044.22644800 4052.56744300 4024.61555500
多线程（线程数=32）	-16.63553047	1227.99166700 1221.78599800 1227.94267300 1220.59474800 1267.42412000
AVX指令集	-16.63552856	1149.19782400 1145.32597100 1149.57106600 1145.93188700 1155.85014700
多线程+AVX指令集	-16.63552856	906.46004400 907.53920100 898.88331600 901.60113600 908.23739800

V499K*E2046K（500K5M1）	max_sum	l_timeMs
example	-16.63553047	5099.90584400 5095.76159000 5106.01794700 5062.20126700 5092.76792300
线程（线程数=32）	-16.63553047	2093.81128400 2095.60083200 2079.36313100 2077.74525300 2087.09064600
AVX指令集	-16.63553047	1980.08000200 1829.62015300 1931.51151800 1886.13781500 1987.09971600

V499K*E2046K（500K5M1）	max_sum	l_timeMs
多线程+AVX指令集	-16.63553047	1519.46776900 1511.23047300 1514.32667300 1514.42493100 1535.81987500

V499K*E5000K（500K5M2）	max_sum	l_timeMs
example	-16.63553047	7642.49233200 7637.20666800 7649.82545400 7675.31530900 7653.70753300
线程（线程数=32）	-16.63553047	4243.99370300 4219.72790500 4250.23249200 4254.65739300 4261.74191100
AVX指令集	-16.63553047	3312.74069700 3315.67255200 3309.71036600 3398.71338900 3312.01001400
多线程+AVX指令集	-16.63553047	2807.18007000 2803.43308400 2813.17235900 2900.01848200 2806.59746400

V1023K*E1023K（1M1M）	max_sum	l_timeMs
example	-16.63553047	7050.10557400 7049.99785500 7047.22892700 7048.55785400 7057.19420200

V1023K*E1023K（1M1M）	max_sum	l_timeMs
线程（线程数=32）	-16.63553047	1441.78346500 1455.68151800 1448.84463100 1585.27109700 1455.90897700
AVX指令集	-16.63553047	1531.65464500 1530.82574300 1529.94576000 1691.02722100 1530.20737800
多线程+AVX指令集	-16.63553047	1118.19982900 1115.20308600 1240.71140700 1117.18684900 1120.17939100

V1023K*E5000K（1M5M）	max_sum	l_timeMs
example	-16.63553047	11772.37075900 11825.61065800 11782.26885600 11786.29813500 11784.60519500
线程（线程数=32）	-16.63553047	5250.77408400 5252.68106500 5262.00468700 5396.92350600 5261.41196100
AVX指令集	-16.63553047	4604.15995100 4622.80982500 4608.84480900 6108.62760700 4653.39820700
多线程+AVX指令集	-16.63553047	3975.55715700 3850.98911600 3819.02202300 3839.07428300 3818.52801400

V5000K*E5000K（5M5M）	max_sum	l_timeMs
example	-16.63553047	35344.06342400 35350.27579600 35398.38685200 35337.21748000 35335.18538600
线程（线程数=32）	-16.63553047	8270.53279000 8516.78416000 7493.87809900 7450.31395400 7450.86149800
AVX指令集	-16.63552856	9774.17346000 9408.03425300 8041.73327100 8044.19948000 8093.76837300
多线程+AVX指令集	-16.63552856	5899.62199500 5877.95509000 6068.60559900 5894.41221100 5887.33824000