图计算挑战赛技术报告

1. 引言

1.1 背景介绍

CCF Computility 2024 图计算挑战赛（CCF Computility Graph Challenge，CCF-CGC）是由中国计算机学会主办的一项图计算挑战赛事。该比赛旨在激发学生对图计算的兴趣，培养创新精神，促进图计算领域专业知识和技能的学习。

1.2 任务概述

本次比赛的任务是优化图卷积神经网络（Graph Convolutional Network, GCN）的推理计算，在不损失计算精度的情况下，在 CPU 平台上以尽可能短的时间完成 GCN 推理计算。

1.3 图卷积神经网络简介

GCN 是一种基于图数据的神经网络模型，能够有效捕捉图中的结构和特征信息。由于其简洁的设计和强大的性能，GCN 在交通预测、蛋白质性质预测、推荐系统等多个领域得到了广泛应用。

2. 基本算法介绍

2.1 图卷积神经网络（GCN）

图卷积神经网络（Graph Convolutional Network, GCN）是一种特殊的神经网络结构，专门用于处理图结构数据。GCN 通过卷积操作来捕捉图中的局部特征，从而实现对图数据的有效学习和推理。

2.2 GCN 的推理公式

在 GCN 中，图卷积层的推理过程可以通过以下公式表示：

$$H^{(l+1)} = \alpha(\hat{A}H^{(l)}W^{(l)})$$

H 是第 l 层的输入顶点特征矩阵。

W 是第 l 层的权重矩阵。

A 是归一化后的图邻接矩阵。

α 是激活函数。

通过这个公式，GCN 能够逐层地更新和提取顶点的特征，从而实现对整个图的表示学习。

2.3 归一化邻接矩阵

归一化邻接矩阵的计算公式为：

$$\hat{A} = D^{-0.5} A D^{-0.5}$$

A 是图的邻接矩阵，表示顶点之间的连接关系

D 是度矩阵，是一个对角矩阵，其第 i 个对角元素为顶点 i 的度数

这种归一化操作能够平衡不同顶点的影响，避免度数较大的顶点在计算中占据过多权重，从而提升 GCN 的推理效果。

2.4 激活函数

在 GCN 中，激活函数用于引入非线性，从而增强模型的表达能力。我们采用两种不同的激活函数：

ReLU（Rectified Linear Unit）：用于第一层的激活函数，其定义为：

$$\mathrm{ReLU}(x) = \max(0, x)$$

LogSoftmax：用于第二层的激活函数，其定义为：

$$\mathrm{LogSoftmax}(x_i) = x_i - \log(\textstyle\sum_j e^{x_j})$$

ReLU 能够有效地捕捉输入特征的正向部分，而 LogSoftmax 则用于归一化输出，使其符合概率分布。

2.5 多层 GCN 结构

在本次比赛中，GCN 模型固定由两个图卷积层构成：

第一层：接收输入的顶点特征矩阵，通过图卷积操作生成新的顶点特征矩阵，并应用 ReLU 激活函数。

第二层：接收第一层输出的顶点特征矩阵，通过图卷积操作生成最终的顶点特征矩阵 ，并应用 LogSoftmax 激活函数。

2.6 图卷积层的计算步骤

GCN 的每一层图卷积操作包括以下步骤：

矩阵乘法（XW）：顶点特征矩阵与权重矩阵相乘。

邻接矩阵乘法（AX）：邻接矩阵与顶点特征矩阵相乘，实现信息传播。

激活函数：对输出特征应用激活函数，增强模型非线性。

3. 设计思路和方法

3.1 任务目标

在 CPU 平台上，以尽可能短的时间完成 GCN 推理计算，同时保持 32 位浮点数精度。我们的设计思路主要包括以下几个方面：

数据结构优化：通过更高效的数据结构来减少内存占用和计算复杂度。

计算优化：通过各种优化技术提高计算效率。

并行计算：利用多线程和 SIMD 指令集，最大化 CPU 资源的利用。

3.2 数据结构优化

3.2.1 使用 CSR 格式存储邻接矩阵

在原始实现中，邻接矩阵采用邻接表存储，这种方法在内存使用和访问速度上存在一定的不足。我们采用 CSR（Compressed Sparse Row，压缩稀疏行）格式来存储邻接矩阵，CSR 格式具有以下优势：

内存高效：CSR 格式只存储非零元素，减少了内存占用。

快速访问：CSR 格式允许快速遍历每个顶点的邻居，提高了计算效率。

CSR 格式的三个数组如下：

csr_val：存储非零元素的值。

csr_col：存储每个非零元素对应的列索引。

csr_row：存储每一行的起始位置索引。

3.3 计算优化

3.3.1 矩阵分块

将大矩阵分块处理可以减少缓存未命中，提高数据访问速度。我们在实现过程中，将矩阵乘法和邻接矩阵乘法的计算进行分块处理，以提高计算效率。

3.3.2 缓存优化

通过适当的数据布局和访问模式优化，可以减少缓存未命中次数，提高缓存命中率。例如，在矩阵乘法和邻接矩阵乘法过程中，尽量按照行优先的顺序访问数据。

3.3.3 SIMD 向量化

SIMD（Single Instruction, Multiple Data）指令可以在单条指令中处理多个数据。我们利用 AVX512 指令集对矩阵运算进行向量化处理，极大地提高了计算速度。

3.4 并行计算

3.4.1 多线程技术

利用多线程可以将计算任务分配到多个 CPU 核心上并行执行，从而提高计算效率。我们使用 pthread 库实现多线程处理，将顶点特征矩阵和邻接矩阵的计算任务分配到多个线程

中执行。

3.4.2 SIMD 向量化

利用 SIMD 指令可以进一步提升计算效率。我们在矩阵乘法和邻接矩阵乘法中使用 AVX512 指令，对数据进行向量化处理。

3.5 详细设计与实现

3.5.1 矩阵乘法（XW）

为了提高矩阵乘法的计算效率，我们采用了多线程和向量化技术。每个线程负责处理一部分顶点的特征，并通过 SIMD 指令加速计算。

3.5.2 邻接矩阵乘法（AX）

在邻接矩阵乘法过程中，我们首先将 CSR 格式的邻接矩阵进行归一化，然后利用多线程和向量化技术进行计算。每个线程负责处理一部分顶点的特征，并通过 SIMD 指令加速计算。

3.5.3 激活函数

我们对 ReLU 和 LogSoftmax 激活函数也进行了多线程优化。每个线程负责处理一部分顶点的特征，从而提高计算效率。

3.6 优化效果

通过上述优化方法，我们显著提升了 GCN 推理计算的效率。具体的优化效果将在实验结果与分析部分详细介绍。

4. 算法优化

在 GCN 推理计算中，我们通过多种优化方法来提高计算效率，包括数据结构优化、计算优化和并行计算。这些优化方法相辅相成，使得我们能够在 CPU 平台上高效地完成 GCN 推理计算。以下是我们具体的优化方法和实现。

4.1 数据结构优化

4.1.1 使用 CSR 格式存储邻接矩阵

CSR（Compressed Sparse Row）格式是一种高效的稀疏矩阵存储格式。相比于邻接表，CSR 格式能够更加高效地利用内存，并且在进行矩阵运算时具有更好的缓存局部性。CSR 格式由以下三个数组组成：

csr_val：存储非零元素的值。
csr_col：存储每个非零元素对应的列索引。
csr_row：存储每一行的起始位置索引。

4.2 计算优化

4.2.1 矩阵分块

将大矩阵分块处理可以减少缓存未命中次数，提高数据访问速度。在矩阵乘法（XW）和邻接矩阵乘法（AX）过程中，我们将大矩阵分成小块进行处理，以提高计算效率。

4.2.2 缓存优化

通过优化数据布局和访问模式，可以减少缓存未命中次数，提高缓存命中率。在实现中，我们尽量按照行优先的顺序访问数据，并使用对齐的内存分配来提高缓存效率。

4.2.3 SIMD 向量化

SIMD（Single Instruction, Multiple Data）指令允许在单条指令中处理多个数据。我们利用 AVX512 指令集对矩阵运算进行向量化处理，极大地提高了计算速度。

4.3 并行计算

4.3.1 多线程技术

利用多线程技术可以将计算任务分配到多个 CPU 核心上并行执行，从而提高计算效率。我们使用 pthread 库实现多线程处理，将顶点特征矩阵和邻接矩阵的计算任务分配到多个线程中执行。

4.3.2 SIMD 向量化

在利用多线程的基础上，我们还使用了 SIMD 指令进行向量化处理，以进一步提升计算效率。在矩阵乘法（XW）和邻接矩阵乘法（AX）中，我们利用 AVX512 指令对数据进行向量化处理。

4.4 整体优化效果

通过数据结构优化、计算优化和并行计算的结合，我们显著提升了 GCN 推理计算的效率。具体的优化效果将在实验结果与分析部分详细介绍。

5. 详细算法设计与实现

在本次 GCN 推理计算中，我们通过多种优化方法提高了计算效率。以下是详细的算法设计与实现，包括每个步骤的具体实现细节。

5.1 第一层图卷积层

5.1.1 矩阵乘法（XW）

第一层图卷积层首先进行顶点特征矩阵和权重矩阵的乘法计算。为了提高计算效率，我们采用了多线程和 SIMD 向量化技术。

5.1.2 邻接矩阵乘法（AX）

在进行邻接矩阵乘法时，我们首先将 CSR 格式的邻接矩阵进行归一化处理，然后利用多线程和 SIMD 向量化技术进行计算。

5.1.3 激活函数（ReLU）

我们对 ReLU 激活函数进行了多线程优化，每个线程负责处理一部分顶点的特征，从而提高计算效率。

5.2 第二层图卷积层

5.2.1 矩阵乘法（XW）

第二层图卷积层的矩阵乘法过程与第一层类似，同样采用多线程和 SIMD 向量化技术，此外采用了转置矩阵方式，将矩阵的按行查找改成按列查找。

5.2.2 邻接矩阵乘法（AX）

第二层的邻接矩阵乘法过程与第一层类似，同样采用多线程和 SIMD 向量化技术。

5.2.3 激活函数（LogSoftmax）

LogSoftmax 激活函数的实现同样采用多线程优化。每个线程负责处理一部分顶点的特征，并对其进行归一化处理。

6. 程序代码模块说明

在本次 GCN 推理计算的实现中，我们将程序划分为多个模块，以提高代码的可读性和可维护性。以下是各个模块的详细说明。

6.1 主函数模块

主函数模块负责整体程序的控制流程，包括读取输入数据、初始化变量、调用各个计算模块和输出结果。

主函数代码：

```
int main(int argc, char **argv) {
    F0 = atoi(argv[1]);
    F1 = atoi(argv[2]);
    F2 = atoi(argv[3]);
    readGraphAndBuildCSR(argv[4]);
    readFloat(argv[5], X0, v_num * F0);
    readFloat(argv[6], W1, F0 * F1);
    readFloat(argv[7], W2, F1 * F2);
    initFloat(X1, v_num * F1);
    initFloat(X1_inter, v_num * F1);
    initFloat(X2, v_num * F2);
    initFloat(X2_inter, v_num * F2);
    TimePoint start = chrono::steady_clock::now();
    XW(F0, F1, X0, X1_inter, W1);
    AX(F1, X1_inter, X1);
    ReLU(F1, X1);
    XW(F1, F2, X1, X2_inter, W2);
    AX(F2, X2_inter, X2);
    LogSoftmax(F2, X2);
    float max_sum = MaxRowSum(X2, F2);
    TimePoint end = chrono::steady_clock::now();
    chrono::duration<double> l_durationSec = end - start;
    double l_timeMs = l_durationSec.count() * 1e3;
    printf("%.8f\n", max_sum);
    printf("%.8lf\n", l_timeMs);
    freeFloats();
    return 0;
}
```

6.2 数据处理模块

数据处理模块负责读取和预处理图数据和顶点特征。我们使用 CSR 格式存储邻接矩阵，并提供函数读取浮点数数组和初始化浮点数数组。

```
extern int v_num;
extern int e_num;
extern std::vector<float> csr_val;
extern std::vector<int> csr_col;
extern std::vector<int> csr_row;
extern float *X0, *W1, *W2, *X1, *X1_inter, *X2, *X2_inter;
void readGraphAndBuildCSR(char *fname);
```

```
void readFloat(char *fname, float *&dst, int num);
void initFloat(float *&dst, int num); void freeFloats();
```

图卷积层模块实现了 GCN 的具体计算，包括矩阵乘法（XW）、邻接矩阵乘法（AX）和激活函数（ReLU 和 LogSoftmax）。

```
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W);
void AX(int dim, float *in_X, float *out_X);
void ReLU(int dim, float *X);
void LogSoftmax(int dim, float *X);
float MaxRowSum(float *X, int dim);
```

并行计算模块实现了多线程计算，包括线程数据结构和多线程处理函数。

```
struct ThreadData {
    int start; int end;
    int in_dim;
    int out_dim;
    float *in_X;
    float *out_X;
    float *W; int dim;
};
void *XW_thread(void *arg);
void *AX_thread(void *arg);
void *ReLU_thread(void *arg);
void *LogSoftmax_thread(void *arg);
```

7. 详细程序代码编译说明

在本次 GCN 推理计算的实现中，我们使用了多个优化编译选项来提高代码的执行效率。以下是详细的编译说明，包括所需的依赖库和具体的编译命令。

7.1 依赖库

在编译和运行程序之前，确保系统已安装以下依赖库和工具：

g++ 编译器：支持 C++11 及以上标准。

pthread 库：用于多线程支持。

AVX 指令集支持：用于 SIMD 向量化。

7.2 编译选项

我们使用了以下编译选项来优化代码执行效率：

-O2：启用基本的优化选项。

-march=native：生成与当前处理器架构相关的代码。

-funroll-loops：展开循环以减少循环开销。

-flto：启用链接时优化（Link Time Optimization）。

-ffast-math：启用快速数学运算优化。

-mavx：启用 AVX 指令集支持。

7.3 编译命令

在终端中执行以下命令进行编译：

g++ -O2 -march=native -funroll-loops -flto -ffast-math -mavx gcn.cpp -o ../example.exe

即可编译成功。

8. 详细代码运行使用说明

在成功编译代码后，接下来是运行程序以完成 GCN 推理计算。以下是详细的运行说明，包括运行命令、输入参数解释、以及示例运行。

8.1 运行命令

在终端中执行以下命令运行程序：

./example.exe <F0> <F1> <F2> <input_graph_file> <input_feature_file> <input_weight1_file> <input_weight2_file>

8.2 参数说明

<F0>：输入顶点特征矩阵的维度（整数）。

<F1>：第一层图卷积输出特征矩阵的维度（整数）。

<F2>：第二层图卷积输出特征矩阵的维度（整数）。

<input_graph_file>：图结构文件，包含图的邻接关系（文件路径）。

<input_feature_file>：顶点特征文件，包含初始顶点特征矩阵（文件路径）。

<input_weight1_file>：第一层权重矩阵文件（文件路径）。

<input_weight2_file>：第二层权重矩阵文件（文件路径）。

8.3 示例运行

我们有以下输入文件：

1024_example_graph.txt：图结构文件。

1024.bin：顶点特征文件。

W_64_16.bin：第一层权重矩阵文件。

W_16_8.bin：第二层权重矩阵文件。

并且参数如下：

F0 = 64：输入顶点特征矩阵的维度为 64。

F1 = 16：第一层图卷积输出特征矩阵的维度为 16。

F2 = 8：第二层图卷积输出特征矩阵的维度为 8。

则运行命令为：

./example.exe 64 16 8 graph/1024_example_graph.txt embedding/1024.bin weight/W_64_16.bin weight/W_16_8.bin

8.4 输出结果

运行程序后，输出结果将包括：
最大行和（max row sum）：用于结果验证。
计算时间（单位：毫秒）：用于评估算法性能。

9. 实验结果与分析

在本次实验中，我们对优化后的 GCN 推理计算进行了详细的测试和分析，以验证我们的优化方法的有效性。实验使用了不同规模的图数据集，并比较了优化前后的性能差异。

9.1 实验环境

处理器：Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
内存：256GB
操作系统：Ubuntu 18.04.6
编译器：g++ 7.5.0
编译命令：g++ -O2 -march=native -funroll-loops -flto -ffast-math -lpthread -mavx2 gcn.cpp -o ../example.exe

9.2 数据集描述

我们使用了 RMAT 生成的随机图数据集，数据集的规模和特点如下：

数据集	顶点数	边数	特征维度 F0	输出维度 F1	输出维度 F2
Large	5000000	5000000	64	16	8
Mid	10000	100000	64	16	8
Small	1024	1024	64	16	8

9.3 实验结果

我们对每个数据集进行了多次运行，取平均值作为最终结果。以下是优化前后的性能对比：

数据集	优化前时间（ms）	优化后时间（ms）	提升比例
Large	29389.44	574.69	51.14
Mid	152.78	6.56	23.29
Small	15.63	0.45	34.73

9.4 性能分析

9.4.1 优化前后的运行时间对比

通过对比优化前后的运行时间，我们可以看到，优化后的 GCN 推理计算在所有数据集上都有显著的提升，特别是在大规模数据集上，提升比例达到了 50 倍。这主要得益于以下几个方面的优化：

数据结构优化：使用 CSR 格式存储邻接矩阵，大大减少了内存占用和数据访问开销。

计算优化：通过矩阵分块、缓存优化和 SIMD 向量化技术，提高了矩阵运算的效率。

并行计算：利用多线程技术将计算任务分配到多个 CPU 核心上并行执行，最大化 CPU 资源的利用率。

9.4.2 内存使用分析

通过数据结构优化，我们使用 CSR 格式存储邻接矩阵，相比于邻接表存储方法，内存使用效率显著提高。在大规模数据集上，CSR 格式的内存使用量大约是邻接表的 50%。

9.5 实验结果总结

通过上述优化，我们显著提升了 GCN 推理计算的效率，特别是在大规模数据集上，优化效果尤为明显。优化后的 GCN 推理计算不仅在运行时间上大幅缩短，同时在内存使用效率上也有显著提升。

本次实验验证了我们设计思路和优化方法的有效性，为进一步研究和优化 GCN 推理计算提供了坚实的基础。