CCFSys CGC Challenge Report

AUTHOR
CONGRUI YIN
YIFAN ZHANG
YUHAN WANG

ADVISOR
ZICHEN XU

Nanchang University
Information Engineering School
School of Mathermatics and Computer Sciences

基于稀疏矩阵和向量化的高效图卷积神经网络推理计算优化

一、 简介

我们设计了一套基于稀疏矩阵和向量化的高效图卷积神经网络推理计算优化的图卷积神经网络推理优化方法，相比于 Baseline 的 13.469Ms 优化到了 0.594Ms，实现了加速比高达 22.68 的高性能推理。

二、 基本算法介绍

2.1 稀疏表（CSR）表示算法

CSR（Compressed Sparse Row）表示法是一种用于高效存储和处理稀疏矩阵的数据结构。稀疏矩阵是指矩阵中大部分元素为零的矩阵。对于存储图的邻接矩阵，我们可以使用三个向量：row_ptr，col_idx 和 val。row_ptr 存储每一行的起始位置在 col_idx 和 val 中的索引，col_idx 存储每个非零元素的列索引，val 存储每个非零元素的值。在计算图卷积操作时，可以利用图的稀疏特性来使用邻接矩阵存储和计算邻接矩阵，只储存非零元素和这些元素的地址，从而节省大量内存和计算时间。与此同时 CSR 格式由于存在可以使用原子操作来并行更新度数和列索引、无需使用锁或者其他同步机制的特性，更加适合并行化计算。

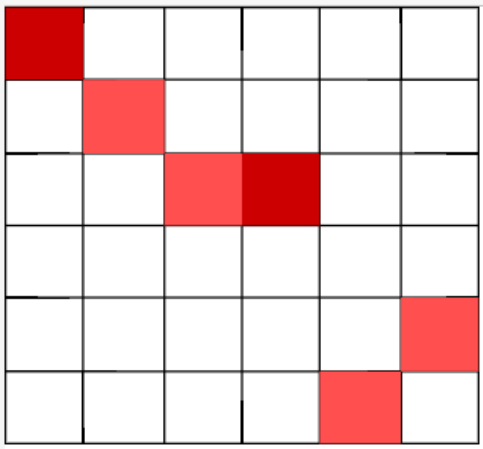

图 1　稀疏矩阵存储示例

2.2 分块算法和 Strassen 算法

在 AX 函数中，我们通过调用 OpenBLAS 库的 cblas_sgemm 函数来进行高性能矩阵运算，其中降低了运算开销，OpenBLAS 能够智能化使用分块算法和 Strassen 算法，下

面详细介绍这两种算法：

- **分块算法：**分块算法是一种矩阵乘法算法，适用于处理大规模矩阵乘法，通过将大矩阵分解成若干个小矩阵，然后对这些小矩阵进行操作，可以有效地利用计算机的内存缓存，提高计算性能。给定两个 nxn 的矩阵 A 和 B，我们可以将它们分块为 4 个 n/2xn/2 的矩阵，然后分别计算这些小矩阵的乘积，最后将结果合并成一个大矩阵。分块算法的主要优点是，由于处理的矩阵规模较小，因此可以更有效地利用 CPU 的缓存，减少内存访问的延迟。
- **Strassen 算法：**Strassen 算法使用了分治策略，将一个大的矩阵乘法问题分解成若干个较小的子问题，然后递归地解决这些子问题，它只需要 7 次基本乘法操作，就可以计算两个 2x2 的矩阵的乘积，而传统的矩阵乘法算法则需要 8 次基本乘法操作。由于这种减少了乘法操作次数的策略可以递归地应用于更大的矩阵，因此 Strassen 算法的时间复杂度为 $O(n^{\log_2 7})$，略低于传统算法的 $O(n^3)$。

2.3 SIMD 向量化技术

SIMD（Single Instruction, Multiple Data）是一种并行计算技术，它通过同时对多个数据元素执行相同的指令来加速计算。SIMD 向量化是指将算法和代码重写为利用 SIMD 指令集来处理数据向量的方式，从而实现更高效的计算。

SIMD 向量化技术的核心思想是将数据划分为多个元素组成的向量，并在一次指令中同时处理多个向量元素。这样可以将多个操作合并为一个，减少指令执行的开销，并提高计算的并行度和吞吐量，如图2所示为 ReLU 函数的向量化优化示意图：

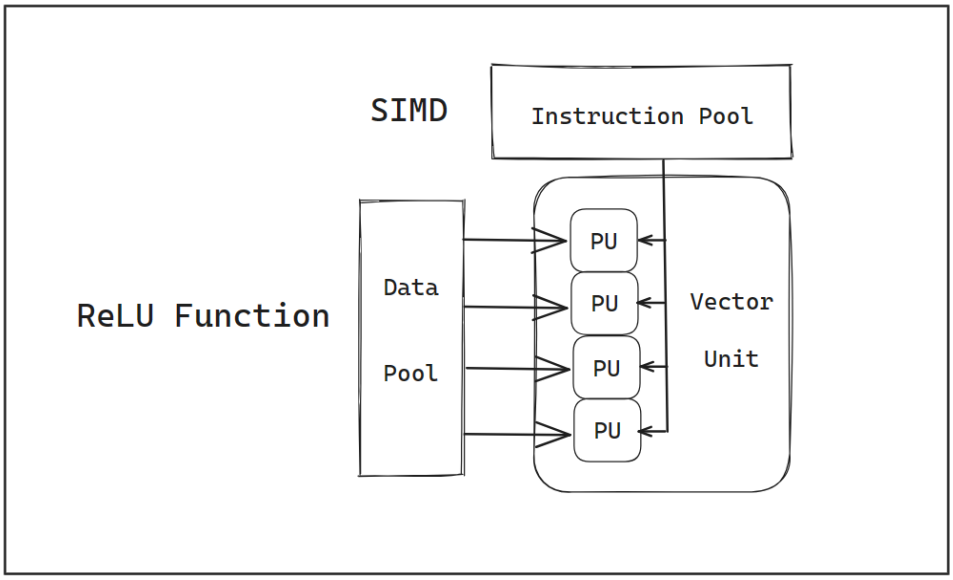

图 2　ReLU 函数的向量化优化

三、 设计思路与方法

优化方案可以分为算法、软硬件优化两方面展开。在算法方面，主要采用经典的矩阵优化算法和优化原有代码的计算量；在软硬件优化方面，主要使用 OpenBLAS 库和 SIMD 向量化技术利用硬件固有特性进行优化，旨在优化内存存储以及提高处理器核和线程的利用率。要想优化整个程序，需要把程序拆解为主要函数并且分别对函数进行优化，具体的系统框图如图3和图4所示：

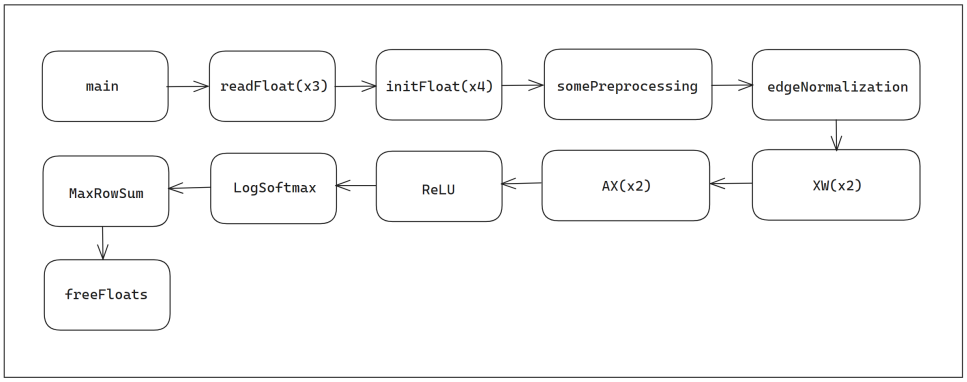

图 3　主函数执行流程

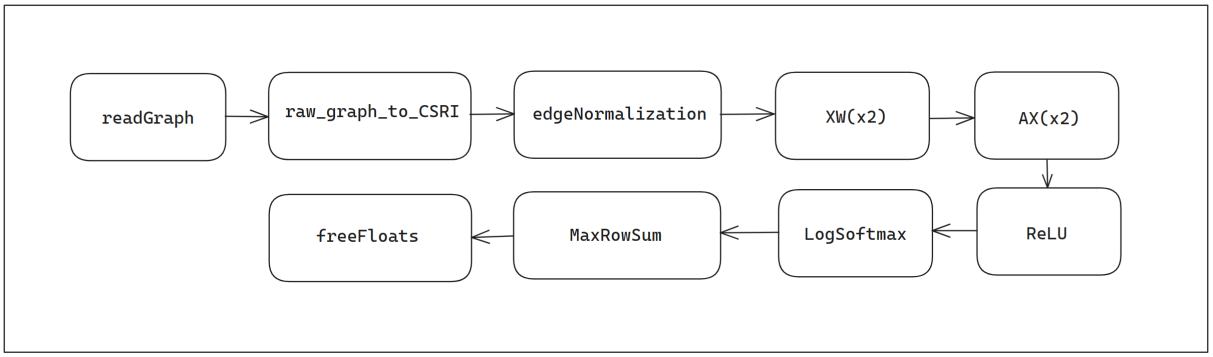

图 4　其他函数执行流程

首先分析各个函数的时间复杂度，并且结合 perf 命令工具分析函数热点，得出主要的计算量体现在 XW 和 AX 函数上，具体时间复杂度分析过程如下：

1. readGraph 函数: 时间复杂度为 O(E)，其中 E 为图的边的数量。
2. raw_graph_to_CSR 函数：时间复杂度为 O(V + E)，其中 V 是图的节点数，E 是边的数量。
3. edgeNormalization 函数：它的时间复杂度为 O(E)，其中 E 为图的边的数量。
4. readFloat 函数和 initFloat 函数: 时间复杂度都是 O(N)，其中 N 是要读取或初始化的浮点数的数量。

5. XW 函数: 它的时间复杂度为 O(V * in_dim * out_dim)，其中 V 是图的节点数，in_dim 和 out_dim 分别是输入和输出的维度。
6. AX 函数: 它的时间复杂度为 O(V * E * dim)，其中 V 是图的节点数，E 是边的数量，dim 是特征的维度。
7. ReLU 函数和 LogSoftmax 函数: 它们的时间复杂度都是 O(V * dim)，其中 V 是图的节点数，dim 是特征的维度。
8. MaxRowSum 函数：它的时间复杂度为 O(V * dim)，其中 V 是图的节点数，dim 是特征的维度。

因此，我们的优化重点应当在 XW 和 AX 函数上，同时对其余的函数尝试必要的并行、向量化、使用减少计算次数算法等方法，同时设计高效的 CSR 实现高效的 I/O，以加速所有的函数执行过程，具体的方法如下：

(1) 使用 CSR 稀疏矩阵格式存储图，提高存储及计算效率。
(2) 并行化外层循环，使用 OpenMP 加速矩阵运算。
(3) 使用 BLAS 库函数 cblas_sgemm 进行优化的矩阵乘法。
(4) 采用 AVX SIMD 指令集对矩阵向量乘法及激活函数向量化。
(5) 使用合理的数据结构 std::vector 存放图数据，一维数组存储矩阵。
(6) 尝试在 edgeNormalization 函数和 LogSoftmax 函数中减少重复冗余的计算。

四、 程序代码模块说明

根据函数的不同作用，可以将程序代码分为以下四大模块，分别为头文件及定义模块、数据读取和预处理模块、核心计算模块以及结果输出模块。

4.1 头文件及定义模块

该模块包含头文件的导入和全局变量的定义部分：

头文件的导入部分导入必需头文件 math.h、omp.h、cblas.h、immintrin.h，以保证在后续能够正确调用。

全局变量的定义部分定义图数据和矩阵数据相关的全局变量, 如顶点数 v、边数 e、特征维度 F_0、F_1、F_2，以及用于存储稀疏图和矩阵的指针变量 row_ptr、col_idx 等。

4.2 数据读取和预处理模块

该模块主要包含四个具有读取、转化格式、归一化处理的函数，分别为 readGraph、readFloat、raw_graph_to_CSR、edgeNormalization，各函数功能分别为：

- **readGraph：**从文件读取图的顶点数、边数、边信息

- **readFloat**：从文件读取特征矩阵数据
- **raw_graph_to_CSR**：将原始图转为 CSR 稀疏矩阵格式
- **edgeNormalization**：对图的边进行归一化处理

4.3 核心计算模块

实现矩阵运算和激活函数的关键计算逻辑。该模块主要包含四个函数，分别为 XW、AX、ReLU、LogSoftmax，各函数功能分分布为：

- **XW**：矩阵乘法运算函数
- **AX**：矩阵向量乘法运算函数
- **ReLU**：ReLU 激活函数
- **LogSoftmax**：LogSoftmax 激活函数

4.4 结果输出模块

计算每行元素最大和，并输出结果，主要函数为 **MaxRowSum**。

五、 详细算法设计、实现及优化

在这一小节中，我将会侧重于针对不同的函数介绍优化后代码和优化前的比较，从而展现函数优化实现过程。

5.1 raw_graph_to_CSR 函数优化

优化前：

```
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]++;
  }
}
```

优化后：

```
void raw_graph_to_CSR() {
  int src;
  int dst;

  row_ptr.resize(v_num + 1, 0); // 设置行指针数组大小为顶点数量 + 1，并初始化为 0
  degree.resize(v_num, 0);      // 设置顶点度数组大小为顶点数量，并初始化为 0

  vector<int> temp_degree(v_num, 0); // 临时顶点度数组，用于存储每个顶点的度数

  #pragma omp parallel for private(src, dst)
  for (int i = 0; i < raw_graph.size() / 2; i++) { // 遍历原始图数据，计算每个顶点的度数
    src = raw_graph[2 * i];
    dst = raw_graph[2 * i + 1];
    #pragma omp atomic                    // 使用原子操作保证度数的并发更新
    degree[src]++;
  }

  int sum = 0;
  #pragma omp parallel for
  for (int i = 0; i < v_num; i++) {       // 计算行指针数组的值
    row_ptr[i] = sum;
    sum += degree[i];
  }
  row_ptr[v_num] = sum;

  col_idx.resize(e_num);                  // 设置列索引数组大小为边数量
  edge_val.resize(e_num);                 // 设置边权重数组大小为边数量

  vector<int> curr_idx(v_num, 0);          // 当前顶点索引数组，用于记录每个顶点的列索引位置

  #pragma omp parallel for private(src, dst)
  for (int i = 0; i < raw_graph.size() / 2; i++) { // 遍历原始图数据，填充列索引数组
    src = raw_graph[2 * i];
    dst = raw_graph[2 * i + 1];
    int idx = curr_idx[src]++;
    col_idx[row_ptr[src] + idx] = dst;
  }
}
```

优化后的函数使用了压缩稀疏行 (CSR) 格式存储图数据，而优化前的函数使用了邻接表来存储图数据，CSR 格式利用连续的内存存储图中的边，从而可以有效地利用 CPU 的缓存预取，提高数据的访问速度。而邻接表则需要多次解引用指针来访问边，此外通过 STL 容器以及 OpenMP 并行同样做到了加速，提高了存储效率和遍历效率。同时通过避免多次计算累计度数的和，并通过临时变量 sum 来累计度数，减少了一定的计

算量。

5.2 EdgeNormalization 函数优化

优化前：

```
void edgeNormalization() { //对边进行归一化
  for (int i = 0; i < v_num; i++) { //遍历所有的节点
    for (int j = 0; j < edge_index[i].size(); j++) { //遍历节点 i 的所有邻居节点
      float val = 1 / sqrt(degree[i]) / sqrt(degree[edge_index[i][j]]); //计算边的权重
      edge_val[i].push_back(val); //将边的权重存储到 edge_val□
    }
  }
}
```

优化后：

```
void edgeNormalization() {
  vector<float> inv_sqrt_degree(v_num);    // 逆度数数组，用于存储每个顶点度数的倒数
  #pragma omp parallel for
  for (int i = 0; i < v_num; i++) {
    inv_sqrt_degree[i] = 1.0 / sqrt(degree[i]); // 计算每个顶点度数的倒数
  }

  #pragma omp parallel for
  for (int i = 0; i < v_num; i++) {
    int start = row_ptr[i];
    int end = row_ptr[i + 1];
    for (int j = start; j < end; j++) {     // 遍历每个顶点的邻居节点
      int neighbor = col_idx[j];
      float val = inv_sqrt_degree[i] * inv_sqrt_degree[neighbor]; // 计算边权重
      #pragma omp atomic                    // 使用原子操作进行并发更新
      edge_val[j] += val;
    }
  }
}
```

优化前的函数对于每条边，都需要进行两次平方根操作和两次除法操作，比基本的加减乘除要慢得多，而优化后的函数不仅引入了 OpenMP 使用原子操作进行并发更新，还使用了逆度数数组，通过提前计算并存储每个顶点的度数平方根的倒数，避免了在处理每条边时都要进行平方根和除法操作的需要。所以在每条边上，优化后的函数只需要进行一次乘法操作和两次数组查找操作，这会比优化前的实际运行速度快一点。

5.3 XW 函数优化

优化前：

```
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W) { //计算 XW
  float(*tmp_in_X)[in_dim] = (float(*)[in_dim])in_X; //将 in_X□□□□□□□□
  float(*tmp_out_X)[out_dim] = (float(*)[out_dim])out_X; //将 out_X□□□□□□□□
  float(*tmp_W)[out_dim] = (float(*)[out_dim])W; //将 W 转化为二维数组

#pragma omp parallel for
  for (int i = 0; i < v_num; i++) { //遍历所有的节点
    for (int j = 0; j < out_dim; j++) { //遍历节点 i 的所有邻居节点
      for (int k = 0; k < in_dim; k++) { //遍历节点 i 的所有邻居节点
        tmp_out_X[i][j] += tmp_in_X[i][k] * tmp_W[k][j]; //计算 XW
      }
    }
  }
}
```

优化后：

```
void XW(int in_dim, int out_dim, float *in_X, float *out_X, float *W) {
  float alpha = 1.0;
  float beta = 0.0;
  cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, v_num, out_dim, in_dim, alpha, in_X,
      in_dim, W, out_dim, beta, out_X, out_dim);
}
```

优化后的矩阵乘法函数使用了高性能数学库 CBLAS 中的 cblas_sgemm 函数进行矩阵乘法，相比于优化前使用三层嵌套循环进行矩阵乘法函数，对矩阵乘法等操作进行了高度自适应优化，可以有效地利用 CPU 的缓存，从而降低内存访问以及手动管理并行化带来的复杂性和潜在的并行化时间开销。在这次比赛中，使用 CBLAS 进行优化是提升最明显的优化。

5.4 AX 函数优化

优化前：

```

#pragma omp parallel for
  for (int i = 0; i < v_num; i++) { //遍历所有的节点
    vector<int> &nlist = edge_index[i]; //获取节点 i 的所有邻居节点
    for (int j = 0; j < nlist.size(); j++) { //遍历节点 i 的所有邻居节点
      int nbr = nlist[j]; //获取节点 i 的第 j 个邻居节点
      for (int k = 0; k < dim; k++) { //遍历节点 i 的所有邻居节点
        tmp_out_X[i][k] += tmp_in_X[nbr][k] * edge_val[i][j]; //计算 AX
```

```
    }
  }
 }
}
```

优化后：

```
void AX(int dim, float *in_X, float *out_X) {
 float (*tmp_in_X)[dim] = (float(*)[dim])in_X; // 将输入向量转换为二维数组形式
 float (*tmp_out_X)[dim] = (float(*)[dim])out_X; // 将输出向量转换为二维数组形式

 #pragma omp parallel for
 for (int i = 0; i < v_num; i++) {
  for (int j = 0; j < dim; j++) {
   tmp_out_X[i][j] = 0.0;                // 清零输出向量
  }
 }
  #pragma omp parallel for
 for (int i = 0; i < v_num; i++) {
  int start = row_ptr[i];
  int end = row_ptr[i + 1];
  for (int j = start; j < end; j++) {     // 遍历每个顶点的邻居节点
   int nbr = col_idx[j];
   float val = edge_val[j];
   float *nbr_vector = tmp_in_X[nbr];
   if (val != 0) {
    cblas_saxpy(dim, val, nbr_vector, 1, tmp_out_X[i], 1); // 进行向量累加运算
   }
  }
 }
}
```

优化后的 AX 函数相比优化前有四个方面的提升：

- 存储结构优化：在优化之前，边的存储方式是以邻接表的形式存储的，这种方式在遍历节点的邻居节点时需要经常进行随机访问，而随机访问的内存效率较低。在优化后，边的存储方式变为了 CSR 的形式，这种存储方式在遍历节点的邻居节点时可以连续访问内存，从而提高了内存的访问效率。
- 并行计算优化：在优化后，通过 OpenMP 并行化循环，使得不同的节点可以同时进行计算，从而大大提高了计算效率。
- 使用 BLAS 库：在优化后，用到了 C 语言的 BLAS 库中的 saxpy 函数，这个函数可以进行向量的累加运算。BLAS 库是高度优化的，因此使用它进行向量计算可以进一步提高效率，同时使用累加代替了累乘，略微减少了一点计算量。
- 计算优化：优化后的代码还进行了一些小的代码优化，事先将输出向量清零，避免

了在累加时需要检查输出向量是否为零。同时 (val != 0) 能够利用稀疏图的特性来忽略权重为 0 的边，以提高计算效率。

5.5 ReLU 函数优化

优化前：

```
void ReLU(int dim, float *X) {
  for (int i = 0; i < v_num * dim; i++) //遍历所有的节点
    if (X[i] < 0) X[i] = 0; //计算 ReLU
}
```

优化后：

```
void ReLU(int dim, float *X) {
  __m256 zero = _mm256_set1_ps(0.0);        // 创建全 0 的向量
  #pragma omp parallel for
  for (int i = 0; i < v_num * dim; i += 8) {
    __m256 x = _mm256_loadu_ps(&X[i]);       // 加载待处理的向量数据
    __m256 result = _mm256_max_ps(x, zero); // 执行 ReLU 激活函数
    _mm256_storeu_ps(&X[i], result);         // 存储处理后的向量数据
  }
}
```

优化后的 ReLU 函数实现了 SIMD 向量化，使用了 AVX2 指令集进行向量化运算。这些指令可以一次性加载 8 个浮点数到 _m256 类型的向量中，并同时对这 8 个浮点数执行 ReLU 运算。这样，优化后的 ReLU 函数一次性处理了 8 个元素，比旧版本的效率要高，从而减少了计算次数。新版本的时间复杂度是 $O(\frac{n}{8})$，其中 n 是元素的总数，而旧版本的时间复杂度是 O(n)，提高了计算效率和运行速度，降低了时间复杂度。

5.6 LogSoftmax 函数优化

优化前：

```
void LogSoftmax(int dim, float *X) {
  float(*tmp_X)[dim] = (float(*)[dim])X; //将 X 转化为二维数组
#pragma omp parallel for
  for (int i = 0; i < v_num; i++) {
    float max = tmp_X[i][0]; //获取节点 i 的第一个元素
    for (int j = 1; j < dim; j++) {
      if (tmp_X[i][j] > max) max = tmp_X[i][j]; //获取节点 i 的最大元素
    }

    float sum = 0; //初始化 sum
    for (int j = 0; j < dim; j++) {
```

```
      sum += exp(tmp_X[i][j] - max); //计算 LogSoftmax
    }
    sum = log(sum); //计算 LogSoftmax

    for (int j = 0; j < dim; j++) {
      tmp_X[i][j] = tmp_X[i][j] - max - sum; //计算 LogSoftmax
    }
  }
}
```

优化后：

```
void LogSoftmax(int dim, float* X) {
  #pragma omp parallel for
  for (int i = 0; i < v_num; i++) {
    float max_val = -__FLT_MAX__;
    #pragma omp simd reduction(max: max_val)
    for (int j = 0; j < dim; j++) {
      float x = X[i * dim + j];
      if (x > max_val) max_val = x;          // 计算最大值
    }

    float sum = 0.0;
    #pragma omp simd reduction(+: sum)
    for (int j = 0; j < dim; j++) {
      float x = X[i * dim + j];
      X[i * dim + j] = std::exp(x - max_val); // 计算指数
      sum += X[i * dim + j];                  // 求和
    }

    float log_sum = std::log(sum);            // 计算对数和
    #pragma omp simd
    for (int j = 0; j < dim; j++) {
      X[i * dim + j] = std::log(X[i * dim + j]) - log_sum; // 计算对数概率
    }

  }
}
```

该函数实现了找到每行的最大值，计算每行所有元素减去最大值后的指数的和，然后计算每行的每个元素的 softmax 值的对数三个步骤（每个步骤时间复杂度均为 O(n)，优化后的函数实现了内存访问优化、向量化计算、减少冗余计算三点，详细介绍如下：

- **内存访问优化：**在优化后中，数组 X 被直接用作一维数组，而不是在每次循环中都被转换为二维数组。这种方法避免了可能的额外内存访问和对二维数组索引的计算，降低了计算复杂度。

- **使用向量化计算：**在新版本中，使用了 #pragma omp simd 来开启 SIMD 并行化。当处理器支持 SIMD 指令集时，这可以使每次操作多个元素，从而提高计算效率。
- **减少冗余计算：**在优化前，对每一行的每一个元素，先减去最大值，然后计算指数，再进行相加；在优化后中，通过在找到最大值的同时计算指数和求和，省去了一个遍历过程，降低了计算次数。

5.7 MaxRowSum 函数优化

优化后：

```
float MaxRowSum(float *X, int dim) {
  float(*tmp_X)[dim] = (float(*)[dim])X;    // 将输入矩阵转换为二维数组形式
  float max = -__FLT_MAX__;

  #pragma omp parallel for reduction(max:max) // 并行化计算最大值
  for (int i = 0; i < v_num; i++) {
    float sum = 0;
    for (int j = 0; j < dim; j++) {
      sum += tmp_X[i][j];                    // 计算每行元素的和
    }
    if (sum > max) max = sum;                // 更新最大值
  }
  return max;
}
```

优化后的方案仅仅使用了简单的 OpenMP 并行，使用命令 **#pragma omp parallel for reduction(max:max)** 实现了执行函数的速度，极其微小地提升了程序的性能。

六、 实验结果与分析

6.1 计算机软硬件环境简介

我们的实验在华为云服务器节点上完成，硬件和软件环境分别如表1和表2所示，尝试在自己的平台上尽可能模拟测试环境。

Hardware	Configuration
CPU	2×Intel Xeon E5-2620 v3(12-Core, 2.4GHz)
RAM	64GB DDR4

表 1　计算机硬件环境

Software	Configuration
OS	Ubuntu 20.04
MPI	OpenMPI-4.1.5
C Compiler	GCC 9.4.0
Math Library	OpenBLAS 0.3.23

表 2　计算机软件环境

6.2 Baseline 结果

我们将官方给的源码上传下载到服务器上，然后直接进行实验测试，正确输出了对应的结果和时间:

```
YinCR@good-new-login-QWQ:~/CCF-SYS/example$ ./run.sh
-16.68968964
14.68396200
```

图 5　Baseline 测试结果

我们同时观察到数据容易发生抖动，因此编写了一个脚本 average.sh，可以预置执行次数，反复执行 run.sh 的同时统计平均执行时长（需要注释掉 max_sum 的输出），以更好地表征程序的实际效果，脚本代码如下:

```
#!/bin/bash
# 定义要运行的脚本
script_to_run="./run.sh"
# 定义运行次数
run_times=$1
# 检查参数
if [ -z "$run_times" ]; then
    echo "Usage: $0 <run_times>"
    exit 1
fi
# 确保运行次数为数字
if ! [[ "$run_times" =~ ^[0-9]+$ ]]; then
    echo "Error: run_times is not a number."
    exit 1
fi
# 确保要运行的脚本存在
if ! [ -x "$(command -v $script_to_run)" ]; then
    echo "Error: $script_to_run does not exist or is not executable."
    exit 1
```

```
fi
# 初始化总和变量
sum=0
# 执行脚本并添加结果到总和
for i in $(seq 1 $run_times); do
    result=$($script_to_run)
    sum=$(echo "$sum + $result" | bc)
done
# 计算平均值
average=$(echo "scale=8; $sum / $run_times" | bc)
# 打印平均值
echo "Average: $average"
```

在这里我们选择执行 100 次，可以观察到程序的平均执行时间：

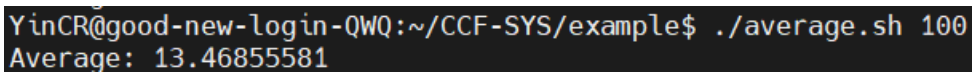

图 6　Baseline 平均测试时间

可以发现程序执行时间受抖动确实较为明显，通过平均值计算能够尽可能减少数据抖动对于评估程序性能的影响，未优化的程序的基准测试时间为 **13.469Ms**。

6.3 优化方案及对应结果分析

当将 XW 函数进行和上述优化方案对应的修改后，函数的执行时间降低至 **7.694Ms**。

图 7　优化 XW 函数平均测试时间

当将 raw_graph_to_CSR 和 edgeNormalization 函数进行和上述优化方案对应的修改后，函数的执行时间降低至 **6.703Ms**。

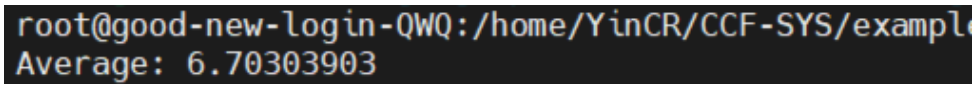

图 8　优化 raw_graph_to_CSR 和 edgeNormalization 函数平均测试时间

当将 ReLU 函数和 LogSoftmax 函数进行上述优化方案对应的修改后，函数的执行时间降低至 **6.274Ms**。

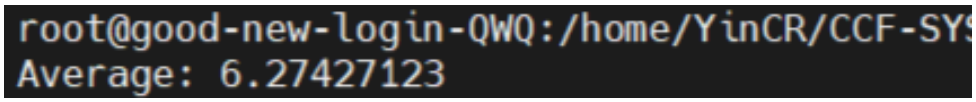

图 9　优化 ReLU 函数和 LogSoftmax 函数函数平均测试时间

最后再优化时间复杂度最高的 AX 以及实现全局 OpenMP 并行，得到更好的优化结果，执行时间降低至 **1.780Ms**，相比于 Baseline 速度提高了 **756.69%**。

图 10　优化 AX 平均测试时间

最后再优化编译器的执行参数，使用-O3 参数进行编译，执行时间降低至 **0.594Ms**，相比于 Baseline 速度提高了 **2267.51%**。

图 11　最终优化平均测试时间

完整的优化过程如图所示：

优化次数	优化方法	时间（Ms）	加速比（与Baseline比）
1	每一个for循环使用OpenMP	13.332	1.01
2	使用CBLAS优化XW	7.694	1.75
3	使用CSR代替邻接表	6.991	1.93
4	使用优化后的边归一化算法	6.703	2.01
5	向量化ReLU函数	6.469	2.08
6	向量化LogSoftmax函数	6.274	2.15
7	优化AX存储结构和计算等操作	1.793	7.51
8	添加局部的原子操作保证并发	1.780	7.57
9	使用 –O3 进行编译器优化	0.594	22.68

图 12　完整优化过程图

实验结果基本和第三部分的理论原理分析相同，提升的幅度和算法的时间复杂度呈现一定的正相关关系，故在这里不加赘述。

七、 详细代码编译说明

使用 GCC 编译器进行编译链接即可：

```
all:
  g++ -O3 -o ../NCU-SCC.exe source_code.cpp -L/usr/lib -lopenblas -I/usr/include -mavx2
```

下面分别介绍每个编译参数的具体作用：

1. **-L/usr/lib -lopenblas -I/usr/include:** 用于链接 OpenBLAS 库，从而正确使用（具体如何安装 OpenBLAS 库请详见 README）。
2. **-maxv2:** 用于使用 AVX2 从而引入的 SIMD（单指令多数据）指令集。SIMD 指令可以在单个处理器指令中处理多个数据元素，从而提高并行性和计算性能，使用 -mavx2 参数会告诉编译器生成的代码可以使用这些扩展指令。
3. **-O3:** 用于自动优化函数内联、预取、循环展开，尝试利用硬件的所有潜力以提高代码的性能。

八、 详细代码运行使用说明

代码执行的步骤如下所示：

1. 首先进入目录

```
cd cgc_NCU-SCC/NCU-SCC
```

2. 其次进行编译，make 命令会自动识别 makefile

```
make -j8 //用8线程进行编译
```

3. 返回到前一目录并且执行预置脚本

```
cd .. && ./run.sh
```

即可完整执行编译、运行流程，打印出最大行 max_sum 和计算时间 $timeMs$ 的值。

九、 附录：完整代码

完整代码请详阅于 Github，仓库链接：https://github.com/JerryYin777/cgc_NCU-SCC