2024 图计算挑战赛技术文档——gogogo 队伍

一、基本算法介绍

图卷积网络（Graph Convolutional Networks，简称 GCN）是一类应用于图结构数据的深度学习模型。GCN 能够在图结构数据上进行节点分类、链接预测、图分类等任务，具有广泛的应用场景，如社交网络分析、推荐系统、化学分子属性预测等。
本体钟，使用了两个卷积层和两个激活函数。包含的步骤有：图的存储、矩阵的乘法、激活函数的运算。所以我们队伍打算从上面的步骤中，逐一结合函数特点进行优化。

二、设计思路和方法

图的存储：GCN 所涉及的图为稀疏矩阵。原有的图结构为邻接表，其访问效率低下，所以我们采用 CSR 结构来存储图。在转换为 CSR 时，利用原子操作来并行更新参数，更加适用于 CSR 并行计算的特点。
矩阵乘法：
这部分包含 XW 和 AX 的运算。首先，从矩阵的最基础的运算来说，我们首先对矩阵进行转置，使得在内存访问上更加连续。接下来，对于矩阵中的主体的循环部分，我们使用 OpenMP 进行并行化处理 for 循环。另外，基于以上几点，加上 SIMD 优化，调用 AVX 和 AVX512 指令集使得矩阵运算优化得更加全面。
激活函数：
使用 OpenMP 并行计算和 SIMD 指令集来加速计算。
结果求解：
使用 OpenMP 并行计算和 SIMD 指令集来加速运算。

三、算法优化

CSR 解决图存储的效率
CSR 格式非常适合进行矩阵-向量乘法操作。在进行这种操作时，只需要遍历非零元素并进行相应的乘法运算和累加操作。这种操作的时间复杂度与非零元素的数量成正比，而不是与矩阵的维度成正比，因此在处理大规模稀疏矩阵时具有明显的效率优势。

OpenMP 解决并行计算
OpenMP（Open Multi-Processing）是一种用于多平台共享内存多处理器编程的应用程序接口（API）。它提供了一组编译器指令、库例程和环境变量，用于开发并行应用程序。OpenMP 的主要好处包括：
1. 易于使用和学习
OpenMP 的设计目标之一就是简化并行编程。通过在代码中添加简单的编译器指令（如 #pragma omp parallel），程序员可以迅速将顺序代码转换为并行代码。与其他并行编程模型相比，OpenMP 的学习曲线较低，使用方便。
OpenMP 提供了丰富的并行构造，支持多种并行模式，例如：

- 并行区域 (#pragma omp parallel)：并行执行代码块。
- 并行循环 (#pragma omp for)：并行化循环迭代。
- 任务并行 (#pragma omp task)：定义并行执行的任务。
- 工作共享 (#pragma omp sections)：将不同代码块分配给不同线程执行。

这种灵活性允许程序员根据具体的应用需求选择最合适的并行模式。

SIMD 利用底层指令集提高运算效率
AVX 通过扩展处理器的向量寄存器和指令集，使其能够在单个指令中处理更多的数据。例如，AVX 指令集

扩展了寄存器的宽度，从 128 位（SSE 指令集）增加到 256 位，甚至在 AVX-512 中增加到 512 位。这意味着处理器可以在同一时刻处理更多的浮点数或整数，从而显著提高了并行计算的性能。

O3 优化

-O3 开启了更高级别的优化技术，能够显著提高生成代码的执行速度。编译器通过更激进的优化策略来减少执行时间，使得最终程序运行得更快。具体的优化策略包括循环展开、函数内联、更高效的指令选择等。高级优化级别使得编译器能够更好地利用处理器的特性，如流水线、寄存器和缓存等。通过优化指令调度和内存访问模式，-O3 可以减少缓存未命中和流水线停顿的情况，从而提高处理器的整体效率。

四、详细算法设计与实现

1. someProcessing

（1）优先计算每个点的度数并存储，使得在计算边权时能够减少计算量，可以直接通过内存地址访问相应数据。

（2）在优化过程中，尽量避免使用 stl 容器，因为其初始化复杂度较一般数组时间复杂度会大幅提高。

（3）使用原子操作__sync_add_and_fetch 和__sync_fetch_and_add，其能够比线程锁的方法速度优于 6-7 倍。

（4）另外同样使用了 OpenMP 进行并行计算的工作。

```
1. void somePreprocessing() {
2.   int raw_size = raw_graph.size();
3.
4.   #pragma omp parallel for schedule(guided)
5.   for(size_t i = 0; i < raw_graph.size(); i+=2) {
6.     __sync_add_and_fetch(&vertex_index[raw_graph[i]+1], 1);
7.   }
8.
9.   size_t sum = 0;
10.  vertex_index[0] = 0;
11.  #pragma omp parallel for schedule(guided)
12.  for(size_t i = 0; i < v_num; ++i) {
13.      sqrt_record[i-1] = sqrt(vertex_index[i]);
14.  }
15.
16.  for(size_t i = 1; i < v_num+1; ++i) {
17.    vertex_index[i] += vertex_index[i-1];
18.  }
19.
20.  #pragma omp parallel for schedule(guided)
21.  for(size_t i = 0; i < raw_size; i+=2) {
22.    int src = raw_graph[i];
23.    int dst = raw_graph[i + 1];
24.    size_t off = __sync_fetch_and_add(&offset[src], 1);
25.    off += vertex_index[src];
```

```
    out_edge[off] = dst;
    edge_val[off] = 1 / (sqrt_record[src] * sqrt_record[dst]);
  }
}
```

2. XW

（1） 转置 W，使得在内存访问时，更加快速。

（2） 使用 AVX512 来机型矩阵 XW 的计算，同时注意处理余项。

```
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;

    float W_T[in_dim*out_dim];

    #pragma omp parallel for
    for(size_t i = 0; i < in_dim*out_dim; ++i) {
        size_t row = i / out_dim;
        size_t colum = i % out_dim;
        W_T[colum*in_dim + row] = W[i];
    }
    float(*tmp_tran_W)[in_dim]=(float(*)[in_dim])W_T;

    #pragma omp parallel for
    for(size_t i = 0;i < v_num;++i)
        for(size_t j = 0;j < out_dim;++j) {
            __m512 c = _mm512_setzero_ps();
            size_t k = 0;
            for(;k+15<in_dim;k+=16) {
                __m512 a = _mm512_loadu_ps((*(tmp_in_X+i)+k));
                __m512 b = _mm512_loadu_ps((*(tmp_tran_W+j)+k));
                c = _mm512_fmadd_ps(a, b, c);
            }
            float cc[16] = {};
            _mm512_storeu_ps(cc, c);
            tmp_out_X[i][j] = cc[0]+cc[1]+cc[2]+cc[3]+c[4]+cc[5]+cc[6]+c
c[7]+cc[8]+cc[9]+cc[10]+cc[11]+cc[12]+cc[13]+cc[14]+cc[15];
        }
    int rem = in_dim & (16 - 1);
    if(rem != 0) {
      #pragma omp parallel for
      for(size_t i = 0;i < v_num;++i)
        for(size_t j = 0;j < out_dim;++j)
```

```
            for(size_t kk = in_dim-rem; kk < in_dim; ++kk)
              tmp_out_X[i][j] += tmp_in_X[i][kk] * tmp_W[kk][j];
    }
}
```

3. AX

(1) 使用 AVX 指令集进行优化

(2) 使用 OpenMP 进行并行计算

```
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;

    int max_threads = omp_get_max_threads();
    const int bs = 16;
    int loop_num = dim/bs;
    __m512 dest_arr[max_threads][loop_num];
    __m512 source_arr[max_threads][loop_num];

    #pragma omp parallel for
    for(size_t v = 0; v < v_num; ++v) {
        int threadnow = omp_get_thread_num();
        for(size_t i = 0; i < loop_num; ++i) {
            dest_arr[threadnow][i] = _mm512_loadu_ps(reinterpret_cast<float const *>(&(tmp_out_X[v][i*bs])));;
        }
        int start = vertex_index[v];
        int end = vertex_index[v+1];
        for(size_t j = start; j < end; ++j)
        {
            int nbr = out_edge[j];
            float weight = edge_val[j];
            __m256 w=_mm256_broadcast_ss(reinterpret_cast<float const *>(&(weight)));
            __m512 w2=_mm512_broadcast_f32x8(w);
            for(size_t i=0;i<loop_num;i+=2){
                __m512 source= _mm512_loadu_ps(reinterpret_cast<float const *>(&(tmp_in_X[nbr][i*bs])));
                dest_arr[threadnow][i] = _mm512_fmadd_ps(source,w2,dest_arr[threadnow][i]);
            }
            for (size_t i = bs*loop_num; i < dim; i++) {
                tmp_out_X[v][i] += tmp_in_X[nbr][i] * weight;
            }
```

```
        }
        for(size_t i = 0; i < loop_num; i++) {
            _mm512_storeu_ps(&(tmp_out_X[v][i*bs]), dest_arr[threadnow][i]);
        }
    }
}
```

4. ReLU

(1) 使用 AVX 指令集进行优化，同时注意处理余项

(2) 使用 OpenMP 进行并行计算

```
void ReLU(int dim, float *X) {
    const int bs = 8;
    int loop_num = v_num * dim / bs;
    __m256 h= _mm256_setzero_ps();

    #pragma omp parallel for schedule(guided)
    for (size_t i = 0; i < loop_num; i++){
        __m256 w = _mm256_loadu_ps(reinterpret_cast<float const *>(& X[i*bs]));
        const __m256 comp2 = _mm256_cmp_ps(w, h, _CMP_LT_OQ);
        const __m256 result = _mm256_blendv_ps(w, h, comp2);
        _mm256_storeu_ps(&(X[i*bs]), result);
    }
    for(size_t i = loop_num*bs; i < v_num * dim; i++)
        if (X[i] < 0) X[i] = 0;
}
```

5. Logsoftmax

(1) 使用 OpenMP 并行计算优化和 GCC 的自动向量优化

```
void LogSoftmax(int dim, float *X) {
    float(*tmp_X)[dim] = (float(*)[dim])X;

    #pragma omp parallel for
    for (size_t i = 0; i < v_num; i++)
    {
        float max = tmp_X[i][0];
        for (size_t j = 1; j < dim; j++) {
            if (tmp_X[i][j] > max) max = tmp_X[i][j];
        }
        float sum = 0;
        #pragma GCC ivdep
```

```
        for (size_t j = 0; j < dim; j++) {
            sum += exp(tmp_X[i][j] - max);
        }
        sum = log(sum);
        for (size_t j = 0; j < dim; j++) {
            tmp_X[i][j] = tmp_X[i][j] - max - sum;
        }
    }
}
```

6. MaxRowSum

(1) 使用 AVX 指令集进行优化

(2) 使用 O 喷 MP 进行并性计算

```
float MaxRowSum(float *X, int dim) {
  float(*tmp_X)[dim] = (float(*)[dim])X;
  float max = -__FLT_MAX__;

  int loop_num = dim/16;

  #pragma omp parallel for schedule(guided) reduction(max:max)
    for (int i = 0; i < v_num; i++) {
        float sum = 0;
        __m512 result =_mm512_setzero_ps();
        for(int j = 0; j < loop_num; j++){
            __m512 a = _mm512_loadu_ps(reinterpret_cast<float const *>(&(tmp_X[i][j*16])));
            result = _mm512_add_ps(result, a);
        }
        sum += ((float *)&result)[0]+((float*)&result)[1]+((float *)&result)[2]+((float*)&result)[3]+((float *)&result)[4]+((float*)&result)[5]+((float *)&result)[6]+((float*)&result)[7]+((float *)&result)[8] + ((float*)&result)[9]+((float *)&result)[10]+((float*)&result)[11]+((float *)&result)[12] + ((float*)&result)[13]+((float *)&result)[14]+((float*)&result)[15];
        for(int j = loop_num * 16; j < dim; j++) {
            sum += tmp_X[i][j];
        }
        if (sum > max) max = sum;
    }
    return max;
}
```

五、实验结果与分析

本比赛测试规模如下：

图规模：

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

环境配置：

CPU: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
CPU(s): 8
Memory: 16G
OS: Ubuntu 22.04.4 LTS
C Compiler: GCC 11.4.0

我们选定**六种规模大小不同的数据集**进行测试，得到了如下的结果。

V(M)*E(M)	0.5*0.5	0.5*1	1*1	0.5*5	1*5	5*5
Example	6013.2214	6677.9538	12534.7763	9226.44404	16083.7684	63050.7184
gogogo	80.339638	100.004784	224.402639	281.054285	380.59501	957.110471
someProcessing	18.0231	32.1351	42.0983	165.423	196.709	302.191
XW1	39.0574	37.2212	111.977	39.8606	73.8739	378.167
AX1	6.17147	9.60356	18.3956	34.0679	47.7179	79.2855
ReLU	0.600894	0.637967	1.53784	0.956962	2.31515	6.16465
XW2	5.13358	4.89066	16.2144	7.05842	10.1553	56.3973
AX2	5.53606	8.47023	17.1131	27.529	37.9756	67.4303
Logsoftmax	5.39302	6.47246	16.034	5.65994	10.9413	63.072
MaxRowSum	0.418	0.566396	1.07411	0.481695	0.899658	4.43522

结果分析：

平均情况下，有 50X 的加速比。其中在 0.5M*5M 数据集的情况下，有 75X 的加速比。
其中对于 AX 的优化效果最为明显，平均能达到 135X。
综合所有数据集，可以看出优化加速效果良好。

程序代码模块说明

所有占用时间的函数模块均使用原有函数接口，没有修改。

详细程序代码编译说明

Makefile 文件代码如下：

```
all:
    g++   -fopenmp -march=native -mavx512f -o ../gogogo.exe source_code.cpp
```

详细代码运行使用说明
1.首先进入项目所在目录：

```
cd cgc_gogogo/gogogo
```

2.其次进行编译

```
make
```

3.退回到前一路径

```
cd ..
```

4.运行文件

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

注：由于源文件中已经包含编译后的程序，所以可以省略第二步和第三步并第一步修改为：

```
cd cgc_gogogo
```