报告文档—HUSTGNN

基本算法介绍

从整体上看，我们在保持原有 GCN 函数接口不变的基础上，改变了原有的图数据结构，并针对函数内部计算进行了优化。

设计思路和方法

1. 对于原有的图数据结构—邻接表，我们将其改造为 CSR 结构，让出边在主存存储结构上更加连续，让计算边上权重 weight 时，Cache 命中率更高。
2. 对于边权重的计算进行了优化，本题中计算公式是一条边的权重起点与终点出度的开平方之积的倒数。而一个顶点的出度开平方的值很容易被重复利用，我们就将其存在数组中，防止重复用时的重复计算。
3. 针对所有的 for 循环进行了 openmp 库的并行优化，针对不同 for 循环进行了不同的调度策略（static、dynamic、guided）。
4. 对于所有涉及可能的矩阵计算进行了 SIMD 优化，使用了 AVX-512 指令集并启用 GCC -O3 选项优化（比赛 CPU 支持）。

算法优化

本文主要使用 for 循环并行计算以及使用 SIMD 优化矩阵乘法来对算法进行优化。

首先优化 for 循环并行计算可以：

1. 提高性能：并行计算允许同时执行多个任务或操作，从而利用多个处理器核心或线程来并行处理数据，加快计算速度。这可以在大规模数据集上显著提高程序的性能，减少计算时间。
2. 加速计算：并行计算允许将任务划分为多个子任务，并同时处理这些子任务。这有助于加速复杂计算，特别是涉及大量迭代或计算密集型的操作。
3. 充分利用硬件资源：现代计算机通常具有多个处理器核心，而并行计算允许充分利用这些核心，同时执行多个任务，从而充分利用硬件资源。
4. 并发性：并行计算可以实现真正的并发性，多个任务可以同时执行而不互相干扰。这在处理实时数据或事件驱动的场景下尤为重要。
5. 可伸缩性：通过使用并行计算，可以实现可伸缩性，即在处理更大规模的数据时，可以通过增加更多的处理器核心或线程来保持高性能
6. 解决复杂问题：并行计算可以帮助解决复杂的计算问题，如大规模数据分析、模拟、图像处理等，这些问题可能在串行计算下难以高效地完成。

另外，SIMD 也可以带来多方面的好处，特别是在涉及大规模矩阵运算时，这种优化可以显著提高性能和效率。下面是一些使用 SIMD 优化矩阵乘法的好处：

1. 提高计算性能：SIMD 指令集允许在一条指令中同时处理多个数据元素，因此可以实现更高效的并行计算。在矩阵乘法中，通过将多个元素同时处理，可以大大加快计算速度，

特别是在大规模矩阵乘法中，性能提升非常显著。

2. 减少指令数：使用 SIMD 指令可以减少需要执行的指令数，从而减少指令级并行性的开销。这使得处理器能够更有效地利用指令级并行性，并提高整体执行效率。
3. 缓存友好性：使用 SIMD 优化可以减少数据传输和缓存访问次数，从而提高缓存友好性。这有助于减少缓存未命中的次数，减少数据访问延迟，提高内存访问效率。
4. 降低功耗：由于 SIMD 指令可以同时处理多个数据元素，可以在一定程度上减少指令执行次数，从而降低功耗和能耗。
5. 平台无关性：SIMD 指令是现代处理器普遍支持的特性，因此使用 SIMD 优化的代码在不同的处理器上都可以获得性能提升，而无需针对特定平台进行优化。

使用 SIMD 优化矩阵乘法可以显著提高计算性能、降低功耗，并且在不同的硬件平台上都能够获得好处，因此是进行大规模矩阵运算时的一个有效的优化手段。

详细算法设计与实现

加载 CSR

1. 并行计算每个点的度数

```
#pragma omp parallel for schedule(guided)
for(size_t i = 0; i < raw_graph.size(); i+=2)
{
  __sync_add_and_fetch(&vertex_index[raw_graph[i]+1], 1);
}
```

图 4.1 加载 CSR-part1

2. 并行计算并保存开方结果

```
#pragma omp parallel for schedule(guided)
for(size_t i = 0; i < v_num; ++i)
{
    sqrt_record[i-1] = sqrt(vertex_index[i]);
}
```

图 4.2 加载 CSR-part2

3. 计算 CSR 索引前缀和

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

图 4.3 加载 CSR-part3

4. 计算权重

```
#pragma omp parallel for schedule(guided)
for(size_t i = 0; i < raw_size; i+=2)
{
  int src = raw_graph[i];
  int dst = raw_graph[i + 1];

  size_t off = __sync_fetch_and_add(&offset[src], 1);
  off += vertex_index[src];
  out_edge[off] = dst;
  edge_val[off] = 1 / (sqrt_record[src] * sqrt_record[dst]);

}
```

图 4.4 加载 CSR-part4

XW

1. 转置 W 矩阵

```
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;
```

图 4.5 XW-part1

2. 计算矩阵相乘结果（使用 AVX-512）

```
#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] + cc[7]
                + cc[8] + cc[9] + cc[10] + cc[11] + cc[12] + cc[13] + cc[14] + cc[15];
    }
}       You, 56分钟前 • first commit
```

图 4.6 XW-part2

3. 计算可能多余部位的矩阵积（该函数的 F0 可能不能整除 16）

```
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];
}
```

图 4.7 XW-part3

AX

1. 使用 AVX、AVX-512 以及并行计算优化矩阵乘法。

```
#pragma omp parallel for
    for(size_t v = 0; v < v_num; ++v)
    {
        int tid = omp_get_thread_num();
        for(size_t i = 0; i < loop_num; ++i)
        {
            dest_arr[tid][i] = _mm512_loadu_ps(reinterpret_cast<float const *>(&(tmp_out_X[v][i*bs])));;
        }
        auto start = vertex_index[v];
        auto 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[tid][i] = _mm512_fmadd_ps(source,w2,dest_arr[tid][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[tid][i]);
        }
    }
```

图 4.8 AX

ReLU

1. 使用 AVX 以及并行计算优化，同时也考虑不能整除 8 的情况进行标量计算。

```
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);
}
// #pragma omp parallel for schedule(guided)
for(size_t i = loop_num*bs; i < v_num * dim; i++)
{
    if (X[i] < 0)
    {
        X[i] = 0;
    }
}
```

图 4.9 ReLU

LogSoftmax

1. 该部分影响较小，使用并行计算以及 GCC 优化向量计算。

```
float(*tmp_X)[dim] = (float(*)[dim])X;
const float MIN_FLOAT =  std::numeric_limits<float>::min();
#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;
    }
}
```

图 4.10 LogSoftmax

MaxRowSum

1. 该部分影响较小，使用 AVX、AVX-512 以及并行计算优化矩阵乘法，同时也考虑不能整除 8 的情况进行标量计算。

```
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;
```

图 4.11 MaxRowSum

实验结果与分析

实验配置

CPU: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
Memory: 187G

数据集

表 5.1 数据集规格统计

Name	V	E
1024_example_graph.txt	1024	4096
1M_5M.txt	1000000	4999999

实验结果

表 5.2 小数据上性能展示

1024_example_graph.txt	Time(ms)	accuracy
Baseline	19.86746800	-16.68968964
HUSTGNN.exe	4.12737300	-16.68968964

表 5.3 大数据上性能展示

1M_5M.txt	Time(ms)	accuracy
Baseline	8196.25330100	-16.68968964
HUSTGNN.exe	90.61282000	-16.68968964

分析

从实验结果上看，我们的优化效果显著，在大数据集情况下能达到 91x 加速比。

程序代码模块说明

我们对函数接口基本没有进行变动，如图 6.1 所示。

1. init_array()函数对用到的数组进行分配内存以及置 0 操作，按本题要求不计入时间。
2. 将原有的 edgeNormalization()函数（计算边 weight）合并在 somePreprocessing()函数中。
3. 其余接口没有进行更改。

```
init_array();        //初始化 不计入时间
// Time point at the start of the computation
TimePoint start = chrono::steady_clock::now();
omp_set_num_threads(omp_get_max_threads());
// Preprocessing time should be included
std::cout<<v_num<<" "<<e_num<<std::endl;
somePreprocessing();

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);

// Time point at the end of the computation
TimePoint end = chrono::steady_clock::now();
chrono::duration<double> l_durationSec = end - start;
double l_timeMs = l_durationSec.count() * 1e3;
```

图 6.1 整体代码模块预览图

详细程序代码编译说明

cd /your_DIR/cgc_HUSTGNN/HUSTGNN
g++ -fopenmp -march=native -mavx512f -o ../HUSTGNN.exe source_code.cpp

或者

cd /your_DIR/cgc_HUSTGNN/HUSTGNN
make

详细代码运行使用说明

以样例数据集为例
cd /your_DIR/cgc_HUSTGNN/
./HUSTGNN.exe 64 16 8 graph/1024_example_graph.txt embedding/1024.bin
weight/W_64_16.bin weight/W_16_8.bin