一、基本算法介绍

GCN的结构是两个卷积层，前面一层使用ReLU激活函数，后面一层使用LogSoftmax激活函数。
ReLU函数： $ReLU(x) = max(0, x)$
LogSoftmax函数： $LogSoftmax(X) = (X_{i,j} - X_{i,max}) - log(\sum_{k=0}^{k=dim-1} e^{X_{i,k}-X_{i,max}})$
单层卷积公式： $X^{(l+1)} = \alpha(AX^lW)$, 其中 A 是归一化后的邻接矩阵，W 是权重矩阵， X^l是输入特征矩阵, $X^{(l+1)}$是输出顶点特征矩阵。

二、设计思路和方法

1. 读取文件
2. 预处理，提前准备好邻接矩阵、归一化的度矩阵
3. 计算XW
4. 计算AX
5. 激活函数 ReLU 和 LogSoftmax
6. 计算最大的顶点特征矩阵行和执行时间

三、算法优化

1. 根据数据所给范围，存在顶点和边数都是 5M的样例，可以判断是一个非常大的图。再结合测试环境CPU物理核数28， 逻辑核数56，内存512G, 这也启发了我使用多线程并行计算的方法来提高计算速度，缩短运行时间。主要是在计算AX和LogSoftmax使用并行计算的方法。

2. 从访问内存方面考虑（减小I/O开销）：二维数组X[row][col]在内存中存储是以行优先的进行存储的(代码中使用的是一维数组), 在访问 X[i * col + j]的时候（即访问一列元素的时候）会有cache命中率低的问题会降低速度，解决方法是将数组沿着对角线进行翻转，即变成访问 X[j * col + i], 这样访问的都是同一行的数据，提高cache的命中率。

3. 在计算XW时，就是普通的矩阵乘法，借助CPU的SIMD指令来进行向量化方法提高矩阵的运算速度，因为SIMD指令可以单条指令操作多个数据，所以一次性先取出2 * k个数据，然后分别相乘再相加，即求出一个值，但是耗时远小于多条指令操作2*k个时间开销小。

四、详细的算法设计和分析以及部分代码模块说明

```
float *transposeMatrix(float *a, int in_dim, int out_dim)
{
  float *b = new float[in_dim * out_dim];

  for (int idx = 0; idx < in_dim * out_dim; ++idx)
  {
    int i = idx / out_dim, j = idx % out_dim;
    b[j * in_dim + i] = a[i * out_dim + j];
  }
  return b;
}
```

上面代码是将矩阵转置，例如A*B, 因为数组是按行存储的，所以让B矩阵转置，这样能够使得每一列处于同一行，提高cache的命中率，减少I/O访问时间。

```
void avx_mul(float *a, float *b_ed, float *c, int v_num, int in_dim, int
out_dim)
{
  float *b = new float[in_dim * out_dim];
  b = transposeMatrix(b, in_dim, out_dim);
  int i, j, k;
  float sum = 0.0;
  float assist = 0.0;
  __m256 r0, r1, r2, r3, r4, r5, r6, r7;
  __m256 c0, c1, c2, c3, c4, c5, c6, c7;
  __m256 avx_mul0, avx_mul1, avx_mul2, avx_mul3,
      avx_mul4, avx_mul5, avx_mul6, avx_mul7;
  __m256 avx_sum0 = _mm256_setzero_ps();
  __m256 avx_sum1 = _mm256_setzero_ps();
  __m256 avx_sum2 = _mm256_setzero_ps();
  __m256 avx_sum3 = _mm256_setzero_ps();
  __m256 avx_sum4 = _mm256_setzero_ps();
  __m256 avx_sum5 = _mm256_setzero_ps();
  __m256 avx_sum6 = _mm256_setzero_ps();
  __m256 avx_sum7 = _mm256_setzero_ps();
  __m256 avx_zero = _mm256_setzero_ps();
  int copy_M = in_dim - in_dim % 64;
  int reserve = in_dim % 64;
  for (i = 0; i < v_num; i++)
  {
    for (j = 0; j < out_dim; j++)
    {
      for (k = 0; k < copy_M; k = k + 64)
      {
        //取a中元素
        r0 = _mm256_loadu_ps(&a[i * in_dim + k]);
        r1 = _mm256_loadu_ps(&a[i * in_dim + k + 8]);
        r2 = _mm256_loadu_ps(&a[i * in_dim + k + 16]);
        r3 = _mm256_loadu_ps(&a[i * in_dim + k + 24]);
        r4 = _mm256_loadu_ps(&a[i * in_dim + k + 32]);
        r5 = _mm256_loadu_ps(&a[i * in_dim + k + 40]);
        r6 = _mm256_loadu_ps(&a[i * in_dim + k + 48]);
        r7 = _mm256_loadu_ps(&a[i * in_dim + k + 56]);
        //取b中元素
        c0 = _mm256_loadu_ps(&b[j * in_dim + k]);
        c1 = _mm256_loadu_ps(&b[j * in_dim + k + 8]);
        c2 = _mm256_loadu_ps(&b[j * in_dim + k + 16]);
        c3 = _mm256_loadu_ps(&b[j * in_dim + k + 24]);
        c4 = _mm256_loadu_ps(&b[j * in_dim + k + 32]);
        c5 = _mm256_loadu_ps(&b[j * in_dim + k + 40]);
        c6 = _mm256_loadu_ps(&b[j * in_dim + k + 48]);
        c7 = _mm256_loadu_ps(&b[j * in_dim + k + 56]);

        avx_mul0 = _mm256_mul_ps(r0, c0);
        avx_mul1 = _mm256_mul_ps(r1, c1);
        avx_mul2 = _mm256_mul_ps(r2, c2);
        avx_mul3 = _mm256_mul_ps(r3, c3);
        avx_mul4 = _mm256_mul_ps(r4, c4);
        avx_mul5 = _mm256_mul_ps(r5, c5);
        avx_mul6 = _mm256_mul_ps(r6, c6);
        avx_mul7 = _mm256_mul_ps(r7, c7);

        avx_sum0 = _mm256_add_ps(avx_sum0, avx_mul0);
```

```
        avx_sum1 = _mm256_add_ps(avx_sum1, avx_mul1);
        avx_sum2 = _mm256_add_ps(avx_sum2, avx_mul2);
        avx_sum3 = _mm256_add_ps(avx_sum3, avx_mul3);
        avx_sum4 = _mm256_add_ps(avx_sum4, avx_mul4);
        avx_sum5 = _mm256_add_ps(avx_sum5, avx_mul5);
        avx_sum6 = _mm256_add_ps(avx_sum6, avx_mul6);
        avx_sum7 = _mm256_add_ps(avx_sum7, avx_mul7);
      }
      // 每次向量乘并求和
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum1);
      avx_sum2 = _mm256_add_ps(avx_sum2, avx_sum3);
      avx_sum4 = _mm256_add_ps(avx_sum4, avx_sum5);
      avx_sum6 = _mm256_add_ps(avx_sum6, avx_sum7);
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum2);
      avx_sum2 = _mm256_add_ps(avx_sum4, avx_sum6);
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum2);
      // 每次求出的c[i,j]
      avx_sum0 = _mm256_hadd_ps(avx_sum0, avx_zero);
      avx_sum0 = _mm256_hadd_ps(avx_sum0, avx_zero);
      assist = avx_sum0[0] + avx_sum0[4];
      c[i * out_dim + j] += assist;
      // 寄存器归0
      avx_sum0 = _mm256_setzero_ps();
      avx_sum1 = _mm256_setzero_ps();
      avx_sum2 = _mm256_setzero_ps();
      avx_sum3 = _mm256_setzero_ps();
      avx_sum4 = _mm256_setzero_ps();
      avx_sum5 = _mm256_setzero_ps();
      avx_sum6 = _mm256_setzero_ps();
      avx_sum7 = _mm256_setzero_ps();
    }
  }
  // 处理剩余的
  assist = 0.0;
  for (i = 0; i < v_num; i++)
  {
    for (j = 0; j < out_dim; j = j + 1)
    {
      for (k = 0; k < reserve; k++)
      {
        assist += a[i * in_dim + copy_M + k] * b[j * in_dim + copy_M + k];
      }
      c[i * out_dim + j] += assist;
      assist = 0.0;
    }
  }
}
```

上面代码是借助CPU的SIMD指令来进行向量化方法来能够通过单条指令控制多组数组的运算，来提高运算速率。

```
void AX(int dim, float *in_X, float *out_X)
{
  int threads = 20;
#pragma omp parallel for num_threads(threads)
```

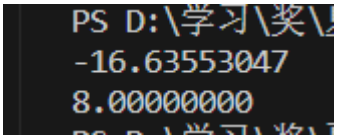
```
    {
      float(*tmp_in_X)[dim] = (float(*)[dim])in_X;
      float(*tmp_out_X)[dim] = (float(*)[dim])out_X;
      for (int i = 0; i < v_num; i++)
      {
        vector<int> &nlist = edge_index[i];
        for (int j = 0; j < nlist.size(); j++)
        {
          int nbr = nlist[j];
          for (int k = 0; k < dim; k++)
          {
            tmp_out_X[i][k] += tmp_in_X[nbr][k] * edge_val[i][j];
          }
        }
      }
    }
  }
```

上面代码使用了多线程并行计算的方式提高计算速度。

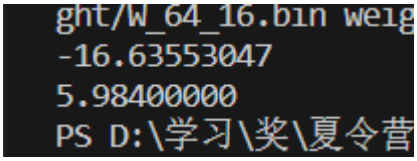
五、结果对比

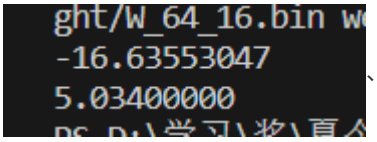
样例数据运行结果优化前后时间对比：

优化前：



优化后：





Ubuntu系统下：

```
carry@ubuntu:~/Desktop/example$ ./carry.out 64 16 8 graph/1024_example_graph.txt embedding/1024.bin wei
ght/W_64_16.bin weight/W_16_8.bin
-16.68968964
5.23334900
carry@ubuntu:~/Desktop/example$ ./carry.out 64 16 8 graph/1024_example_graph.txt embedding/1024.bin wei
ght/W_64_16.bin weight/W_16_8.bin
-16.68968964
5.31990900
```