CUDA-Accelerated Graph Convolutional Neural Network

Yuanhao Jia **Zongze Li**

Hengda Shi **Jintao Jiang**

1. Introduction

Graph Convolutional Network is introduced by Thomas N. Kipf and Max Welling in ICLR 2017. It is a semi-supervised learning approach dealing with Non-Euclidean space graph-structured data. Analysing graph structured data has become more practically important these days, and GCN is a breakthrough in the field of graph embedding network as it achieves much higher accuracy in node and graph classification. It is similar to the Convolutional Neural Network in the sense that it would aggregate neighbor information to node. It is also different from CNN because images are Euclidean-structured data but graphs are not. The relationship between node and its neighbors cannot be easily quantified. Briefly speaking, GCN first calculates *Symmetric Normalized Combinatorial Laplacian* of the graph, and does spectral decomposition via polynomial approximation to find eigenvalues and eigenvectors of the Laplacian. They then derived a graph convolution formula from Fourier Transform where eigenvectors of Laplacian are used as transform basis. Later on, just like traditional CNN, feature vectors are multiplied with graph convolution formula to calculate outcome and total loss. At last, network weights will be updated through backward propagation. The propagation order can be seen in the following diagram:

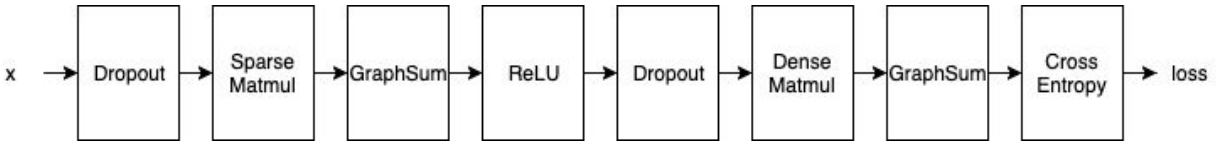


Figure 1.1 GCN layer workflow

Based on the official demo version of GCN, Liwei Cai and Chengze Fan from CMU implemented GCN from scratch in C++ and accelerated it with OpenMP SIMD. We continue their work by accelerating GCN with GPU and rewrite their from-scratch sequential C++ code with CUDA.

Codes are tested and evaluated on AWS g3s.xlarge instance which has Tesla M60 and CUDA Version 10.0.130. The following parts of the report include implementations of each module and its corresponding kernel functions, high level workflow and process of code integration, testing and result evaluating on different datasets, and the final conclusion. Our Github repo is https://github.com/hengdashi/cuda_gcn.

2. Implementation

2.1 Kernel Implementation

2.1.1 Adam Optimizer

The original form of Adam optimizer takes as input a vector of *Variable*, and for each variable, it loops through its parameters and updates them using gradients with exponential decay. There are two for loops,

so in our implementation, we parallelize the inner loop. A thread will know its position in the vector of parameters for a variable. If it is larger than the vector size, it will do nothing. Otherwise it will apply the update logic to the corresponding parameter.

```
__global__
void cuda_Adam_step_kernel(float* grad, float* data, float* m, float* v, bool decay, float
weight_decay, float beta1, float beta2, float eps, float step_size, int varsize) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i >= varsize) return;

    float g = grad[i];
    if (decay) g += weight_decay * data[i];
    m[i] = beta1 * m[i] + (1.0 - beta1) * g;
    v[i] = beta2 * v[i] + (1.0 - beta2) * g * g;
    data[i] -= step_size * m[i] / (sqrtf(v[i]) + eps);
}
```

2.1.2 Sparse Matrix Multiplication

The sparse matrix is compressed using CSR format where there are 3 vectors: indptr, indices and data. *Data* stores all non-zero values in the matrix, *indices* stores the column index of each value, and indptr stores for each row the starting and ending indices of its non-zero values in *data*. The data structure that represents the sparse matrix is shown below.

```
class SparseIndex {
public:
    std::vector<int> indices;
    std::vector<int> indptr;
    void print();
};
```

The original form of SparseMatmul has three for loops. First one iterates through *indptr* for each row, second one iterates through each non-zero value in that row, and third one iterates through each class, which has an independent length, and does the computation. We parallelize the first and third for loop where the block index indicates the row and the thread index indicates the class. Each thread will do the work for the designated row and class.

```
__global__
void cuda_SparseMatmul_forward_kernel(float *a_in, float *b_in, float *c_in, int *indptr,
int *indices, int p) {
    int i = blockIdx.x;
    int k = threadIdx.x;
```

```
    #pragma unroll
    for (int jj = indptr[i]; jj < indptr[i + 1]; jj++) {
        int j = indices[jj];
        c_in[i * p + k] += a_in[jj] * b_in[j * p + k];
    }
}
```

2.1.3 Cross Entropy Loss

The cross entropy loss of GCN is implemented as *CUDACrossEntropyLoss* class. The original implementation uses one for loop to update cross entropy loss of each class of data. When rewriting this part with CUDA, each kernel thread deals with one class of data. The number of elements in each class (*num_classes*) is much smaller (less than ten) so we do not provide parallelism here. As for the total loss, we use thrust::reduce to accumulate all the loss value computed in each thread. Thrust::reduce provides accelerated reduction operation in GPU. It internally applies reduction operation in each pair of two adjacent elements simultaneously. So the whole operation process is like a binary tree. The main part of the kernel function is presented below:

```
__global__
void cuda_CrossEntropy_forward_A_kernel(float* logits_data, float* logits_grad, bool
training, int num_classes, int* truth, int* count, float* thread_loss, int size) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= size) return;
    if (truth[i] < 0) {
        count[i] = 0;
        return;
    }
    float *logit = &logits_data[i * num_classes];
    float max_logit = -1e30, sum_exp = 0;
    #pragma unroll
    for (int j = 0; j < num_classes; j++)
        max_logit = fmax(max_logit, logit[j]);
    #pragma unroll
    for (int j = 0; j < num_classes; j++) {
        logit[j] -= max_logit;
        sum_exp += expf(logit[j]);
    }
    if (training) {
        #pragma unroll
        for (int j = 0; j < num_classes; j++) {
            float prob = expf(logit[j]) / sum_exp;
            logits_grad[i * num_classes + j] = prob;
        }
        logits_grad[i * num_classes + truth[i]] -= 1.0;
    }
    count[i] = 1;
```

```
    thread_loss[i] = logf(sum_exp) - logit[truth[i]];
}
```

The thrust::reduce part of code is presented here:

```
thrust::device_ptr<int> count_ptr = thrust::device_pointer_cast(d_count);
int count = thrust::reduce(count_ptr, count_ptr + logitsPerClass, (int)0,
thrust::plus<int>());
thrust::device_ptr<float> loss_ptr = thrust::device_pointer_cast(d_loss);
*loss = thrust::reduce(loss_ptr, loss_ptr + logitsPerClass, (float)0.0,
thrust::plus<float>());
CUDA_CHECK(cudaGetLastError());
```

2.1.4 GraphSum Module

CUDAGraphSum implements the graph convolution operations. It reads CUDAVariable *layer1_var from *SparseMatmul* module and CUDASparseIndex *graph as our graph-structured data. As we mentioned in Matrix Multiplication module, CUDASparseIndex uses compressed Sparse Row (CSR) format to store sparse matrix represented graph.

In the forward kernel function, d_indptr[dst] and d_indptr[dst+1] indicate one non-zero elements sequence. For each non-zero element, dst indicates column indices of non-zero elements in the i-th row. Based on the following graph convolution formula:

$$h^{(l+1)} = \sigma(\sum_j 1/c_{ij} h^l W^l)$$

Each element in *data is updated with accumulated parameter coef. In order to further accelerate coef computing process, we unroll for loops in each kernel thread. Kernel function of CUDAGraphSum::forward is presented as below:

```
__global__
void cuda_GraphSum_forward_kernel(float *d_in_data, float *d_out_data, int *d_indptr, int
*d_indices, int dim, int numNodes) {
    int src = blockIdx.x;
    int j = threadIdx.x;

    int ptr_src_0 = d_indptr[src];
    int ptr_stc_1 = d_indptr[src + 1];

    #pragma unroll
    for (int i = ptr_src_0; i < ptr_stc_1; i++) {
        int dst = d_indices[i];
        float coef = 1.0 / sqrtf(
                (ptr_stc_1 - ptr_src_0) * (d_indptr[dst + 1] - d_indptr[dst])
        );
```

```
        // This only works for undirected graphs. Should be out[dst] += coef * in[src]]
        d_out_data[src * dim + j] += coef * d_in_data[dst * dim + j];
    }
}
```

2.1.5 ReLU Activation Function

Class *CUDAReLU* implements the ReLU activation function. It receives output from the *GraphSum* module, updates *bool *mask* and *CUDAVariable *layer1_var2*, which will be the input variable for next *Dropout* module. *Bool *mask* is a boolean array which assign zero to data[i] if it is less than zero. The original implementation uses one for loop to update parameters in ReLU. In our implementation, however, each kernel thread deals with one element in *data, *mask, and *grad. The kernel functions of forward propagation is listed here:

```
__global__
void cuda_ReLU_forward_kernel(float *d_in_data, bool *d_mask, const long unsigned int
datasize, bool training) {
   uint i = (blockIdx.x * blockDim.x) + threadIdx.x;
   if (i >= datasize) return;
   bool keep = d_in_data[i] > 0;
   if (training) d_mask[i] = keep;
   if (!keep) d_in_data[i] = 0;
}
```

2.1.6 Matrix Multiplication

The original form of matrix multiplication for A(m * n) * B(n * p) = C(m * p) is that, for each i in m and j in p, we loop k through n and compute C[i][j] = ∑A[i][k] * B[k][j]. Here we accelerate this process using tiling method, which efficiently utilize shared memory to reduce the read/write memory latency compared to global memory. The kernel function of CUDAMatmul::forward is demonstrated as below:

```
__global__
void cuda_Matmul_forward_kernel(const float *a, const float *b, float *c, const uint m,
const uint n, const uint p) {
    __shared__ float tileA[TILE_SIZE][TILE_SIZE];
    __shared__ float tileB[TILE_SIZE][TILE_SIZE];
    int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y;
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;
    int range = (n-1) / TILE_SIZE + 1;
    float res = 0;

    #pragma unroll
    for (int i = 0; i < range; i++) {
```

```
        if (row < m && i * TILE_SIZE + tx < n)
            tileA[ty][tx] = a[row * n + i * TILE_SIZE + tx];
        else
            tileA[ty][tx] = 0;
        if (col < p && i * TILE_SIZE + ty < n)
            tileB[ty][tx] = b[(i * TILE_SIZE + ty) * p + col];
        else
            tileB[ty][tx] = 0;

        __syncthreads();
        #pragma unroll
        for (int j = 0; j < TILE_SIZE; j++)
            res += tileA[ty][j] * tileB[j][tx];
        __syncthreads();
    }
    if (row < m && col < p)
        c[row * p + col] = res;
}
```

2.1.7 Dropout Layer

For each variable, we will loop through its parameters, and decide if to set it to 0 by some random probability. This is a map operation for 1-D array, and thus can be easily parallelized. In our kernel function, each thread knows its position in the array, and will set its value to 0 by the probability.

```
__global__
void cuda_Dropout_forward_kernel(float *in, int *mask, curandState *state, const uint size,
const float p, const float scale, const bool useMask) {
    float x;
    bool keep;
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < size) {
        x = curand_uniform(&state[id % MAX_THREAD_PER_BLOCK]);
        keep = x >= p;
        in[id] *= keep ? scale : 0;
        if (useMask) mask[id] = keep;
    }
}
```

2.2 Integration

Initially, we wrote CUDA malloc and free inside each function as it was easier to collaborate and test independently. However, this would involve a great amount of unnecessary I/O for each function call. When we integrated our code, we decided to put all the data and modules inside GPU in the beginning. So to enforce this, we implemented our own *CUDAVariable* like the following. We copy the data into GPU

when we construct the *CUDAVariable* and free them when we destroy it. All the data will be wrapped into *CUDAVariable*, and modules can only use *CUDAVariable* instead of plain data directly. By doing this, our speedup increased from 2x to 6x.

```
CUDAVariable::CUDAVariable(int size, bool requires_grad) {
    this->requires_grad = requires_grad;
    this->size = size;
    CUDA_CHECK(cudaMalloc((void**) &data, size * sizeof(float)));
    if (requires_grad) {
        CUDA_CHECK(cudaMalloc((void**) &grad, size * sizeof(float)));
    }
}

CUDAVariable::~CUDAVariable() {
    CUDA_CHECK(cudaFree(data));
    if (requires_grad) CUDA_CHECK(cudaFree(grad));
}
```

3. Evaluation

We compare the original C++ from-scratch implementation with our CUDA accelerated implementation. We also compare them with the original implementation from Kipf's Tensorflow CPU version, and Tensorflow GPU version.

3.1 System Configuration

System	GPU	CUDA Version	CPU	Compiler
AWS g3s.xlarge	Nvidia Tesla M60	10.0.130	Intel Xeon CPU E5-2686	GCC

3.2 Dataset

We used four different datasets to test our implementation. These datasets are arranged by increased number of nodes and edges. For Reddit dataset, since it's too large, we only tested it on the C++ sequential code and our CUDA version.

Dataset	Type	Nodes	Edges	Classes	Features
Cora	Citation Network	2,708	5,429	7	1,433
Citeseer	Citation Network	3,327	4,732	6	3,703
Pubmed	Citation Network	19,717	44,338	3	500

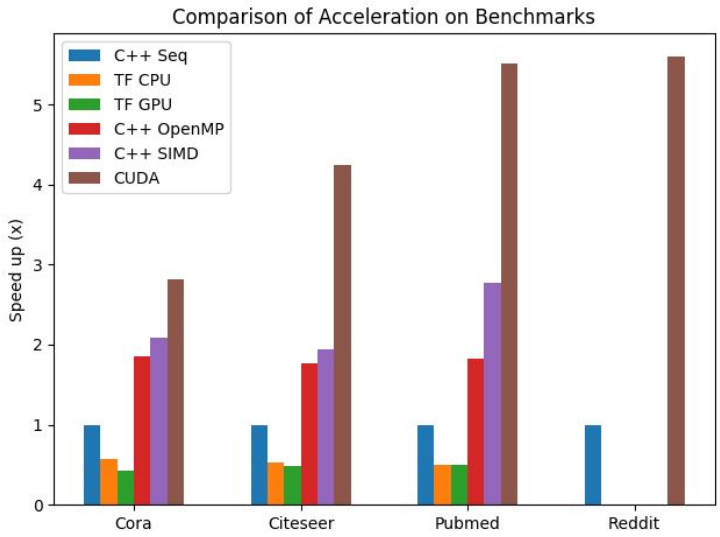
Reddit	Online Post Graph	232,965	11,606,919	41	602

3.3 Raw results

Performance Raw Results (second)				
Method	**Cora**	**Citeseer**	**Pubmed**	**Reddit**
Tensorflow CPU	1.01350	1.70631	12.18792	N/A
Tensorflow GPU	1.39597	1.83618	12.33965	N/A
C++ sequential	0.58667	0.89984	6.07379	595.36652
C++ OpenMP	0.3155	0.5078	3.3382	N/A
C++ OpenMP SIMD	0.2817	0.4649	2.1903	N/A
CUDA	0.20823	0.21186	1.10340	106.23713

3.4 Result

We used the C++ sequential code as our baseline. As we can see from the figure, the C++ sequential version is faster than the TensorFlow GPU and CPU version, which are written in Python. The OpenMP and OpenMP SIMD version are around 2x faster. Our CUDA version is in general 3-6x faster. What's more, as indicated by the figure, when the datasets become larger, we will get a greater benefit from GPU acceleration.



4. Conclusion

To conclude, based on Liwei Cai's from-scratch implementation of GCN, we developed a CUDA accelerated version and rewrite most of their codes. When running with Pubmed dataset, CUDA accelerated version is 11x faster than the original tensorflow implementation, and 2x faster than OpenMP SIMD version implemented by Liwei Cai. Therefore, We conclude that our CUDA-Accelerated GCN is successful.

Reference

Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." *arXiv preprint arXiv:1609.02907* (2016).

Cai Liwei, Fan Chengze, parallel-gcn, (2019), GitHub repository, https://github.com/cai-lw/parallel-gcn