# Implementation of Graph Convolution Network on GPU

Divija Gogineni
University of Texas at Austin
divija@utexas.edu

Sangram Kate
University of Texas at Austin
sangram.kate@utexas.edu

## ABSTRACT

**tasks in recent years, ranging from image classification and video processing to speech recognition and natural language understanding. The data in these tasks are typically represented in the Euclidean space. However, there is an increasing number of applications where data are generated from non-Euclidean domains and are represented as graphs with complex relationships and interdependency between objects. The complexity of graph data has imposed significant challenges on existing machine learning algorithms. Recently, many studies on extending deep learning approaches for graph data have emerged. In this survey, we provide a comprehensive overview of graph neural networks (GNNs) in data mining and machine learning fieldsDeep learning has revolutionized many machine learning . In this article, we provide the implementation details of one such type of graph neural network called graphical convolutional network. Traditionally, it has been observed that the capabilities of GPUs to handle thousands of threads in parallel achieving maximum performance in case of multithreaded programs.Deep Neural networks essentially can achieve better performance with highly parallel architecture, thus are good to be executed over a GPU. We aim to maximize the performance of our network by using Cuda to run the network over a GPU to optimize the performance. This article will explain the underlying details in the implementation of a neural network for graph using c++ and Cuda.**

## CCS CONCEPTS

• Parallelism • Computer architecture • Graph data structure

## KEYWORDS

Graph Convolutional Network, GPU Architecture

## 1 Introduction

Neural Networks have come a long way in the past two decades. Traditionally, these neural networks were implemented only for Euclidean data, but as more and more data is available in non-structured form, this has given rise to Graph Neural Networks. Graph Convolution Network is a variant of GNN in which involves 'convolution' that is multiplying the input nodes with a weight matrix. In CNN, the input neurons are multiplied with a set of weights and this filter acts as a sliding filter across the whole image and the CNN learns features from the neighboring pixels. Similarly, in GCN, the adjacency matrix of the graph is accounted for in the forward pass. The adjacency matrix represents the connection between the nodes and it enables the model to learn the feature representations based on the node connectivity.

Graph neural networks show somewhat common universal architecture. They are called convolutional networks because filter parameters are shared across all locations in the graph. For graph convolutional models each node of a graph, n, has some feature representation associated with it. the node n of graph G is connected to other nodes through edges e. We use the node features along with a neural network to classify the nodes of the graph. In the process we first sum up the features of all neighboring nodes of node n which then ran over the neural network. the representation of graph in terms of adjacency matrix is used to compute the summation of neighboring node's features. the neural network layer can be expressed as following:

$$H^{(l+1)} = f(H^{(l)}, A)$$

Where $H^{(l+1)}$ is the output of node feature summation, $H^{l}$ is the input activation matrix and A is the adjacency matrix.

In this paper, we implement GCN for node classification and propose several optimizations that provide speedup over the existing implementations. To summarise, our contributions are:

1. Find the sweet spot that achieves both good training time and high prediction accuracy.
2. Identify improvements through careful data movement and orchestration.

3. Improve training time of matrix multiplication: Use Nvidia's CuSparse library call to compute Sparse Matrix multiplication(SpMM) and use tiling techniques to speed-up Dense Matrix Multiplication(GeMM).


Figure 1: **Computation stages in Two-layered GCN**

## 2 Neural Network Architecture

In order to implement the Graph Convolution Network, we divided the work into 2 parts. a) collection of data structures in suitable format to provide as an input to the Neural Network. b) Implementation of neural network. We created our own definition of matrix as a data structure format in cuda C++ class to store the data. As we want to utilize the benefits of GPU, we provide the flexibility to each data structure to allocate the memory over Host and GPU. Although matrix is a 2-D data structure, we stored the data into both GPU and Host in 1-D linear array. the storage of data is done over row-major order. The Matrix data structure stores the information of rows and columns of the matrix within it which is further used by different layers for matrix accesses.

To implement the neural network, we created the definition of different layers in the form of cuda c++ class. we aim to use a Convolutional Layer called the Node aggregation layer to convolve the neighboring nodes together forming feature convolution. Linear layer is defined to produce a dense multiplication function where the neurons are fully connected to each other. Rectified linear unit(ReLU) activation function provides non linearity into the neural network. We provide the softmax unit as a layer to produce the output by normalizing the predictions for the given neural network. Each of these layers were defined to achieve maximum parallelism by dividing the workload into maximum possible threads to achieve the benefits of GPU. To understand the deviation from the actual target we need to use a cost function which provides us the degree of misprediction. Gradient Descent on this cost function is used for back propagation on the network to update the parameters.

```
Matrix& NodeAggregator::forward(Matrix& A,bool training,bool freeMatrix){
this->A = A;
Z.allocateMemoryIfNotAllocated(A.shape);
SpMM(nnz_data, row, col, A.data_device, Z.data_device, A.shape.x, nodes, nnz);
A.freeMem();
return Z;
}
```

Figure 2: Implementation of Forward function for Node Aggregation

```
Matrix& NodeAggregator::backprop(Matrix& dZ, float learning_rate) {
this->dZ = dZ;
dA.allocateMemoryIfNotAllocated(dZ.shape);
SpMM(nnz_data, row, col, dZ.data_device, dA.data_device, dZ.shape.y, nodes, nnz);
dZ.freeMem();
return dA;
}
```

Figure 3: Implementation of Backward function for Node Aggregation

### 2.1 Node Aggregation

Node information is aggregated with its neighbours before they are further processed. The intuition behind this is that the network neighborhood defines a computation graph. Dot product of the Adjacency matrix and the node feature matrix represents the sum of the neighboring nodes features. However, this fails to account for the feature vector of the node itself. To overcome this, the adjacency matrix is to be added to its identity matrix. It can be observed from the datasets in Figure 12 that the adjacency matrix of the nodes is very sparse as each node is connected to very few nodes. The adjacency matrix can be compressed using CSR(Compressed Sparse Row) format which needs three vectors to hold the data: Data, ColInd and RowPtr.Data refers to the non-zero values of the matrix, ColInd stores the column index of each value and RowPtr stores for each row the starting and ending indices of its non-zero values. Nvidia's Cusparse library is used to compute the Sparse-Matrix Multiplication(SpMM). The implementation of the forward and backward paths are shown in figures 2 and 3 respectively.

### 2.2 Linear Layer

We implement the dense matrix multiplication layers or fully connected layers as Linear Layers. fully connected layers are essential in Convolutional networks to estimate the feature and classification connectivity using a set of weights and biases. The sets and bias in fully connected layers assign a significant relationship between a feature and classification. the higher the weight associated with a feature the more dependence is observed for classification of nodes by that feature representation. We provide the bias in the relation of input activation and weight as an interceptor which shifts the output to the weighted some closer towards the prediction.

In order to design the linear layer in our implementation we characterize our Linear Layer class with weight matrix, bias matrix and input activation matrix. The construction of a linear layer uses the dimensions as an input to allocate bias and weight matrix of essential dimensions.

We initialize the weight matrix with random values provided from a random distribution as described in figure 2. We populate the bias matrix with zeros to initialize and expect the neural network to update the bias to suitable values through the process of backpropagation.

```
void LinearLayer::initializeWeightsRandomly(){
    std::default_random_engine generator;
    std::normal_distribution<float> normal_distribution(0.0, 1.0);
    for(int x = 0; x < W.shape.x; x++){
        for(int y = 0 ; y < W.shape.y; y++){
            W[x * W.shape.y + y] = normal_distribution(generator) * weights_init_threshold;
        }
    }
    W.copyHostToDevice();
    free(W.data_host);
}

void LinearLayer::initializeBiasWithZeros() {
        cudaMemset(b.data_device, 0, b.shape.x * b.shape.y* sizeof(float));
}
```

Figure 4: initialization of weight and bias matrix in Linear layer.

In the forward computation, the aggregated feature matrix is taken as the input and is multiplied by the weight matrix. In the backward computation, the propagated error is fed in as input and is multiplied by the weight matrix. Weights, bias and activations are updated in the backward computation.
The multiplication of input activations and weights can be seens a matrix multiplication where input activations is a 2 dimensional matrix of input batch and weight matrix assigned for the transformation of each node of a fully connected layer to the output. We utilize the parallelism in the row-column multiplications and create threads based on the combinations of row-column sets. The full implementation of forward and backward functions for fully connected layers are described in figure 5 and 6.

```
__global__ void linearLayerForward( float* W, float* A, float* Z, float* b,
                                                                    int W_x_dim, int W_y_dim,
                                                                    int A_x_dim, int A_y_dim){
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    int Z_x_dim = A_x_dim;
    int Z_y_dim = W_x_dim;

    float Z_value = 0;

    if( row < Z_x_dim && col < Z_y_dim){
       for(int i=0; i< W_y_dim; i=i+1){
          Z_value += W[i + W_y_dim * col] * A[i + A_y_dim * row];
       }
       Z[row * Z_y_dim + col] = Z_value + b[col];
    }
}
```

Figure 5: Forward function for fully connected layers where activations and weights are multiplied as matrix multiplication.

```
__global__ void linearLayerBackprop( float* W, float* dZ, float*dA,
                                                                    int W_x_dim, int W_y_dim,
                                                                    int dZ_x_dim, int dZ_y_dim){
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    int dA_x_dim = dZ_x_dim;
    int dA_y_dim = W_y_dim;

    float dA_value = 0.0f;
        if (row < dA_x_dim && col < dA_y_dim) {
                    for (int i = 0; i < W_x_dim; i++) {
                                dA_value += W[i * W_y_dim + col] * dZ[ i + dZ_y_dim * row];
                    }
                    dA[row * dA_y_dim + col] = dA_value;
          }
}
```

Figure 6: Backpropagation in Linear Layer.

## 2.3 ReLU

The Rectilinear activation unit provides the essential non-linearity within the neural network model. ReLU function is defined as a linearity equation for all positive values while being a constant zero for all the values which are negative. We chose ReLU as our non-linear functionality for it's computational efficiency both in forward and backward direction. ReLU is also very efficient for gradient descent along the minimum cost derivation. It helps the program to converge faster and achieves better performance.

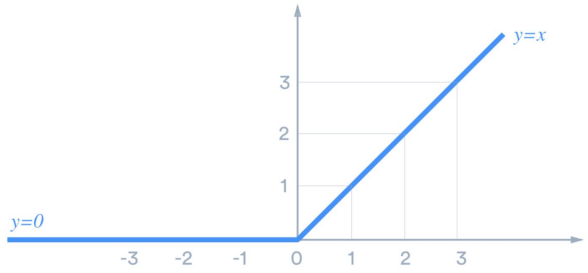


Figure 7: Rectilinear function for Non-linearity within Neural Network.

To Implement the relu function over GPU platform, we needed to compare a value to be positive or negative to generate the output. As each element within a matrix is passed through the ReLU in atomic nature towards the other element of the matrix, all element-wise comparisons can be done in parallel. to achieve the maximum performance from ReLU, we provided a maximum number of blocks and threads to perform ReLU. We implement the backpropagation of Relu Layer in the similar way to that of forward. The gradient of error is traversed to only positive inputs as no negative input provides the contribution towards the result generation.

```
__global__ void ReluActivationForward(float* Z, float* A,float* Stored_Z, int Z_x_dim, int Z_y_dim) {

        int index = blockIdx.x * blockDim.x + threadIdx.x;

        if (index < Z_x_dim * Z_y_dim) {
                A[index] = fmaxf(Z[index], 0);
                Stored_Z[index] = A[index];
        }
}
```

Figure 8: Implementation of Forward function for ReLU.

```
__global__ void ReluActivationBackprop(float* Z, float* dA, float* dZ, int Z_x_dim, int Z_y_dim) {

        int index = blockIdx.x * blockDim.x + threadIdx.x;
        if (index < Z_x_dim * Z_y_dim) {
                if (Z[index] > 0) {
                        dZ[index] = dA[index];
                }
                else {
                        dZ[index] = 0;
                }
        }
}
```

Figure 9: implementation of backward function for ReLU

## 2.4 SoftMax

The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1. The equation for softmax can be expressed as shown in the formula below.

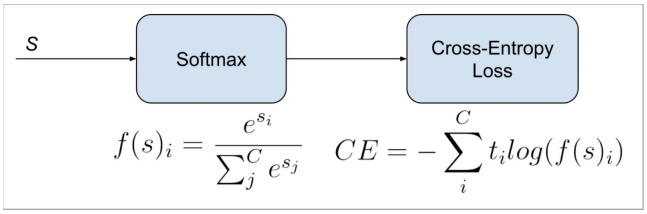


Figure 10: Soft Max equation

Softmax function is ideal in the cases when we need to use the Neural network as a solution for multiclass classification. Our Neural network implementation was targeted to do the same.
In order to implement the SoftMax layer in the neural network, we performed the exponential addition function for input activations to the softmax layer. the maximum parallelism in this layer was limited by the dimensions of input activation. We designed the activation in such a way that we could perform the parallel addition of multiple input activations over multiple threads as shown in figure 11.
We perform the backpropagation of the gradient in the similar way where the optimization is provided by utilizing the parallelism along multiple gradients in a batch. Figure 12 provides the backpropagation implementation for the Softmax Layer.

```
__global__ void SoftMaxForward( float* A, float* Z,int A_x_dim, int A_y_dim){
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int Z_x_dim = A_x_dim;
    int Z_y_dim = A_y_dim;

    float sum = 0.0f;

    if(col << Z_x_dim){
       for(int i=0; i< Z_y_dim; i=i+1){
            float tmp = exp(A[i * Z_x_dim + col]);
            Z[i* Z_x_dim + col] = tmp;
            sum += tmp;
       }
       for(int i= 0; i < Z_y_dim; i=i+1){
            Z[i * Z_x_dim + col] /= sum;
       }
    }
}
```

Figure 11: The Cuda Implementation for computation of softmax forward function.

```
__global__ void SoftMaxBackprop( float* dZ, float*dA, int dZ_x_dim, int dZ_y_dim){
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int dA_x_din = dZ_x_dim;
    int dA_y_din = dZ_y_dim;

    float sum = 0.0f;
        if (col < dA_x_dim) {
            for(int i=0; i< dA_y_dim; i=i+1){
                float tmp = exp(dZ[i * dA_x_dim + col]);
                dA[i* dA_x_dim + col] = tmp;
                sum += tmp;
            }
            for(int i=0; i< dA_y_dim; i=i+1){
                for(int i=0; i< dA_y_dim; i=i+1){
                    if(i==j){
                        dA[j* dA_y_dim + col] += dZ[j * dA_y_dim + i] * (sum - exp(dZ[j * dA_y_dim + i]))/ (sum * sum) * exp(dZ[j * dA_x_dim + i]);
                    }
                    else{
                        dA[j* dA_x_dim + col] -= dZ[j * dA_x_dim + i] *  exp(dZ[j * dA_x_dim + i])/ (sum * sum) * exp(dZ[j * dA_x_dim + i]);
                    }
                }
            }
        }
}
```

Figure 12: Cuda Implementation of Softmax backpropagation function.

## 2.5 Cost Function

To classify each of the nodes of graphs using the features we used one hot encoding of the classes. The output of a graph convolutional network is a one dimensional vector of size equal to the number of classes used for classification. We use binary classification technique where output values are the probabilities of an input node being classified to that class.
To estimate the difference between prediction and target value and train the network towards the higher accurate predictions we use a binary cross entropy value as our cost function described in figure 13.

$$CE = -\sum_{i=1}^{C'=2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1)$$

Figure 13: Binary cross entropy equation

As we implement our network to process input in terms of batches, we use a cost function to calculate the cost of multiple input activations from each node, we do these cost computations all parallel by using single thread for each input activation. the overall computation code is described in figure 14.

```
__global__ void binaryCrossEntropyCost(float* predictions, float* target, int size,int prediction_y, float* cost) {

        int index = blockIdx.x * blockDim.x + threadIdx.x;
        float partial_cost = 0.0f;
        if (index < size) {
            for(int i = 0 ; i < prediction_y; i++){
                partial_cost += target[index* prediction_y + i] * logf(predictions[index * prediction_y + i])
                                                                + (1.0f - target[index * prediction_y + i])
                                                                * logf(1.0f - predictions[index * prediction_y + i]);
            }
            atomicAdd(cost, - partial_cost / prediction_y);
        }
}
```

Figure 14: Binary cross entropy function

## 3 DATASETS

We choose three datasets for evaluating our implementation. The GCN is implemented using Galois[3] which reads the input datasets and outputs the adjacency matrix in CSR format.

| Dataset | Nodes | Edges | Classes | Features |
|---|---|---|---|---|
| Cora | 2708 | 5429 | 7 | 1433 |
| Citeseer | 3327 | 4732 | 6 | 3703 |
| Pubmed | 19717 | 44338 | 3 | 500 |

Figure 12: Datasets used for evaluation

## 4 CHALLENGES

1. During the implementation, we encountered many issues relating to memory leaks in device memory.

2. Galois Library calls were not straightforward for extracting adjacency matrix information.

## 5 REFERENCES


[1] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 752-768

[2] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In ICLR, 2016.

[3] Galois Library. https://github.com/IntelligentSoftwareSystems/Galois

[4] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc.

[5] Deep Graph Library. https:github.com/dmlc/dgl