Parallel GAT Attention Layer

Senyu Tong
Carnegie Mellon University
Pittsburgh, PA 15213
senyut@andrew.cmu.edu

Yile Liu
Carnegie Mellon University
Pittsburgh, PA 15213
yilel@andrew.cmu.edu

1 Summary

In this project, we parallelized the calculation of the attention layer of a Graph Attention Network (GAT). We had a sequential implementation of it in C as the baseline, and then optimized it using OpenMP and CUDA on the GPU, where we reached $10.34\times$ and $283.45\times$ speedups respectively.

2 Background

Graph Attention Network (GAT) Veličković et al. [2017] is a powerful deep learning architecture operates on graph-structured data, which achieves state-of-the-art performance on various classification tasks. It adopts a graph convolution approach in getting the multi-head attention of each node, which is very computationally heavy.

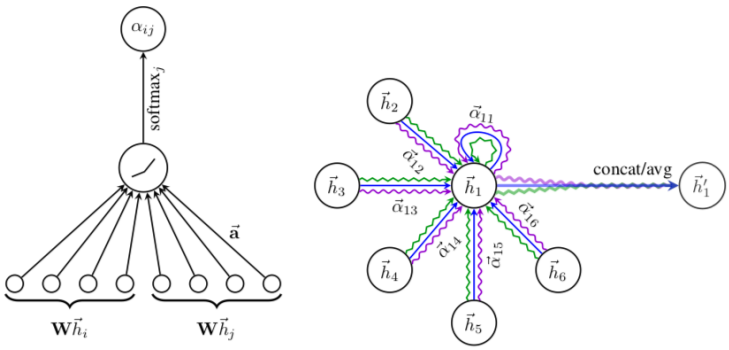

Figure 1: **Left**: The attention mechanism **Right**: An illustration of multi- head attention (with K = 3 heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_t$.

Given a graph with N nodes and m edges, for each node we have node feature $h_i \in R^F$ where F is the dimension of the feature. For each head, we have the two sets of learnable parameters, weights, $W \in R^{F\times F'}$, used for linear activation to transform the dimension of node features to F'; and self-attentions, $a \in R^{2*F'}$, a vector used in calculating attention coefficients for each edge e_{ij}.

Preprint. Work in progress.

Formally, given $\mathbf{h} = \{h_1, h_2, ..., h_N\}, h_i \in R^F$, and K sets of parameters $\{W^1, ..., W^k\}, W^i \in R^{F \times F'}, \{a^1, ..., a^k\}$, we'll output the transformed embedding $\mathbf{h}' = \{h'_1, h'_2, ..., h'_N\}, h'_i \in R^{KF'}$

$$\alpha_{ij} = \frac{exp(lrelu(a^T[Wh_i||Wh_j))}{\sum_{k \in N_i} exp(lrelu(a^T[Wh_i||Wh_k))} \tag{1}$$

$$h'_i = CONCAT_{k=1,...,K} \sigma(\sum_{j \in N_i} \alpha_{ij}^k W^k h_j) \tag{2}$$

We split the process to 3 steps, the first is to compute linear activation for every nodes, then we get attention coefficients on each edge, and finally get the output embedding. This is reflected as a nested 4-layer for loops.

In our C implementation, we have structures for graph and for parameters. For the graph, we represent the adjacency matrix with an int array of length `nnode` + `nedge` and an additional int array of length `nnode` + 1 to indicate the starting index for each adjacency list, and we represent the node features as a 2D double list of length `nnode` $\times$ `nfeature`. For the parameters' structure, for each head, apart from a 2D double list `weights` and a double list `attentions`, we also store the cached features after different forms of activation.

Algorithm 1: Sequential Forward phase of attention layer

INPUT: original graph feature F, weights W, current attentions a, cached features after linear transform L, temporary attentions A, transformed embedding dimension `out_feature`
OUTPUT: updated graph with new features

```
init new embedding F' as a 2D list of shape (nnode, nhead * out_feature);
for H = 0 to nhead - 1 do
    // Step1:  Get linear activation
    for every node i do
        for every out feature j do
            result = 0.0;
            for every input feature k do
                result += F[H][i][k] * W[H][k][j];
            end
            L[H][i][j] = result;
        end
    end
    // Step2:  get attention coefficients
    for every node i do
        left = 0;
        for every out feature j do
            left += a[H][j] * L[H][i][j]
        end
        for every neighboring node j do
            right = 0;
            for every feature k do
                right += a[H][out_feature + k] * L[H][j][k];
            end
            A[H][j] = exp(lrelu(left + right));
        end
        denom = the sum of every element in A[H];
        // Step 3:  get α_ij and new embedding
        for every neighboring node j do
            α_ij = A[H][j] / denom;
            for every feature k do
                F'[i][H * out_feature + k] += α_ij * L[H][j][k];
            end
        end
    end
end
```

3 Approach

After evaluating the speedup possibility, we have adopted two approaches to the optimization, OpenMP and CUDA. Although CUDA seems to be much faster than OpenMP, we still explore OpenMP to accommodate the situation that GPU is not always available and for our own curiosity.

3.1 OpenMP

3.1.1 Work Distribution To Threads

The naive sequential implementation involves one outer loop over the number of heads and two separate inner loops over the number of nodes in the graph. Since the number of heads is usually small, parallel on different heads is not an ideal option and many available threads will be idle. As a result, we explored parallelism on the two inner loops by distributing the iterations to different threads, in which each thread will be in charge of the computation for multiple nodes.

3.1.2 Arithmetic Intensity and Cache Optimization

The first inner loop is matrix multiplication. We changed from the three-loop direct computation to blocked computation to increase the cache locality.

We further divided the second big inner loop to three smaller loops. New algorithm is presented below. To compute the left and right value for each node requires memory read from the same array. Computing the value all at once allows the value to be fully utilized when residing in the cache and increases arithmetic intensity. We further split the computation into two more loops because computing α_{ij} for every node requires memory read from array A that we built from previous loop and we want to fully utilize it when it is still in cache. Without doing so, it may be evicted before we reach next iteration because the next loop requires large amount of memory access especially when the number of $out\_features$ is huge. The third inner loop doesn't have much difference from sequential implementation.

Algorithm 2: Algorithm Adapted for OpenMP

```
initialization;
for every head do
    #pragma omp for;
    multiply feature matrix with weight matrix;
    #pragma omp for;
    for every node do
        Compute left value;
        Compute right value;
        Write the value into array A;
    end
    #pragma omp for schedule(dynamic, 64);
    for every node i do
        Compute α_ij and the sum;
    end
    #pragma omp for schedule(dynamic, 64);
    for every node do
        Compute the new embedding;
    end
end
```

3.1.3 Work Scheduling

We used static scheduling for the first two loops because the amount of computation for every node is the same. We used dynamic scheduling for the latter two for-loops because the amount of computation for every node is dependent on the number of neighbor it has. To avoid the impacts of scheduling overhead, we used granularity 64.

3.1.4 Ineffective Attempts

We have tried to get rid of the large outer-loop over number of heads and merge the computation for different heads into the inner loops. However, it slows down the program. We conjectured that cache performance is the bottleneck here.

3.2 CUDA

3.2.1 Implementation

The CUDA implementation involves 4 rounds launch of GPU computations. They can not be combined because the next computation is dependent on previous one and the mapping to logical threads are different. Notice that we get rid of the outer loop over different heads and each time we launch the CUDA kernel, we will compute the result for all heads. We have modified our algorithm so every step is similar to operating on grids.

- Step 1: **Matrix multiplication.** The output matrix is of dimension $(nnode \times (out \times nhead))$ We use the same blocked implementation as mentioned in recitation. Each thread maps to one entry in the output matrix. Every CUDA block corresponds to a block in the output matrix and every warp is a sequence of horizontal entries.
- Step 2: **Compute the left and right value as mentioned in the sequential algorithm.** We will store the result into a matrix of dimension $nnode \times (2 * nhead)$. Each logical thread corresponds to an entry in the resulting matrix. Each block is of dimension $LBLK \times 2$. We use this dimension so that every block will only do computation for one head. Since the memory referenced for each head is different, we are trying to explore the cache locality. Every warp is a sequence of vertical entries. We didn't use horizontal entries for a warp because here every block only has 2 columns.
- Step 3: **Compute α_{ij}.** Computing α_{ij} involves two step. The first step is for node i, compute value e_{ij} such that j is a neighbor of i, e_{ij} is the numerator of α_{ij} see (1). Then compute $\sum_i e_{ij}$. There will be two output matrix, the first matrix is of dimension $nnode \times (nnode \times nhead)$. For every head, the entry ij will contain value e_{ij} if $A[i, j] = 1$ where A is the adjacency matrix of the graph. Then second matrix is of dimension $nnode \times nhead$, which stores $\sum_j e_{ij}$ for every node in every head. Every logical thread of this CUDA kernel corresponds to one entry in the first matrix. Every block has size $1 \times nnode$, so every block will do computation for one node in one head. After e_{ij} has been computed, we synchronize every thread in the block and do the summation. The algorithm for summation is presented below. The runtime reduces from O(n) to O(logn).

Algorithm 3: Kernel for computing sum of an array

```
for (int offset=nnode/2; offset>=1; offset/=2) do
    if idx<offset then
        relu_sum[i*nnode*nhead+j] = relu_sum[i*nnode*nhead+j] +
          relu_sum[i*nnode*nhead+j+offset];
    end
    __syncthreads();
end
```

- Step 4: **Compute new embedding.** The new embedding is a matrix of dimension $nnode \times (out \times nhead)$. Each logical thread will correspond to one entry in the matrix. Every block has dimension $(1 \times nnode)$, so every block will compute the out features of one node in one head. The warp is sequence of horizontal entries in the matrix. We assign blocks in this way because the compute for each node will require different memory reads.

3.2.2 Cache Optimization

We explored the cache locality by using some block shared memory. For example, in step 1, we first loaded all data from the corresponding block to the block shared memory. In step 2, every thread in one head will read from the same array, so we first load the array into the block shared memory. In step 4, every thread in one block will refer to same row in the matrix we computed in step 3, so we first loaded the row into the block shared memory.

3.2.3 Ineffective Attempts

The original sequential implementation for step 3 is to loop through every node then loop through its every neighbor. Our initial thoughts is to have every logical thread correspond to one node. However, we still need to loop through every one of its neighbor inside each thread, which doesn't fully exploit the parallelism of CUDA. Then we decided to have every logical thread correspond to every node pair. Although it may waste some thread because it doesn't do any work, it plays a role in doing the summation and we feel it should still be faster for relatively dense graph.

4 Results

We benchmark on two graphs. One is regular graph, and the other is just randomly generated graph. Both graphs have 2048 nodes, 820224 edges and 2048 input features. Each node will have 800 edges in the regular graph. The weights are randomly generates and we set the number of out features equal the number of in features.

4.1 OpenMP

	Sequential	2 threads	4 threads	8 threads	Results
Regular Graph	156.317s	55.91s	28.02	14.23s	10.98x
Random Graph	158.732s	58.119s	29.938s	15.354s	10.34x

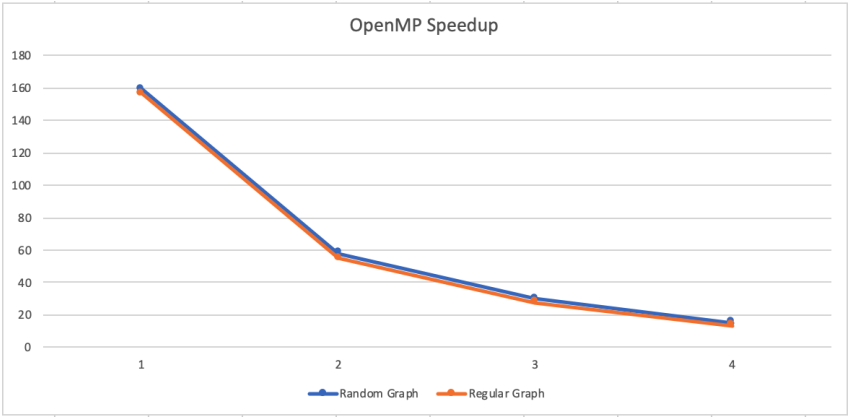

Figure 2: The performance with different number of threads

4.1.1 CUDA

	Sequential	CUDA	Results
Regular Graph	156.317s	0.54s	289.47x
Random Graph	158.732s	0.56s	283.45x

We further test our CUDA implementation on a CORA dataset. The Cora dataset consists of 2708 scientific publications classified into one of seven classes. The citation network consists of 5429 links. We use 4 heads here.

Sequential	CUDA	Results
95.755s	0.63s	152x

4.2 Discussion

We didn't time different parts of our implementation and do comparison because we have changed the order and structure of our implementation, (algorithm is still the same). Timer different parts and do comparison here doesn't make much sense here.

From OpenMP result, we saw that from sequential version to 2 threads, we have achived more than 2x speedup due to both parallelism on multiple threads and arithmetic and cache optimization, which shows that our optimization is effective. Then we see that the speedup is close to linear. We conjecture that since our graph is large, the thread launch overhead doesn't play a big role here. Comparing to random graph, the regular graph demonstrate a speedup closer to linear, however, only slightly. We conclude that our implementation should work well on most of graphs under expectation.

The CUDA implementation achieves a huge speedup, which shows that our optimization is effective. However, comparing to matrix multiplication implemented in recitation, which achieves more than 1000x speedup, there might still be some space for improvement. We conjecture one possible reason is that the process involved many memory copy from host to device, which is unavoidable, and it can significantly slows down the computation. The CUDA implementation achieves relatively equally well on both regular and random graphs, with slightly more improvement on regular graphs. This is the same as our expectation because we have corresponds between blocks and nodes, the imbalance of work might impact the performance here.

We noticed that CUDA will achieve a much greater speedup on dense graphs, as indicated by the CORA dataset which sparse. Because at some stage in our implementation, we map every logical thread to an entry in adjacency list, which means that if the entry is 0, the thread will remain idle for some time because no computation is needed if the two nodes are not neighbor of each other. This is a bottleneck of our implementation and the performance doesn't scale with the number of edges. This will be a focus of our future work.

5 Division of Work

Equal work was performed by both project members.

References

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

6 Project Code Link

https://github.com/Senyu-T/Parallel-Graph-Attention-Network-Forward-Phase