HIGH-PERFORMANCE DATA AND GRAPH ANALYTICS CONTEST

Luca Romanò
Agnese Taluzzi

a.a. 2022/2023

1 Introduction

1.1 Objective

The contest of the High-Performance Graph Analytics course aimed to enhance the performance of a sequential CPU implementation of a Graph Convolutional Network (GCN) in C++ for Semi-Supervised Classification. The goal was to leverage GPU acceleration without using CUDA libraries (e.g., CuBLAS) while maintaining a satisfactory level of accuracy compared to the baseline implementation.

The contest consisted of two main steps:

1. GPU acceleration: Focus on the acceleration of the workload;
2. Model optimization: Focus on exploring various approaches to modify the model or experiment with different hyperparameters to improve the accuracy of the predictions.

1.2 Data

The source code utilizes three citation networks as datasets: Cora, Citeseer, and Pubmed. These datasets represent graph structures with nodes representing documents and edges representing citation links. The specifications of each dataset are shown in Table 1.

Table 1: Dataset Specifications

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

During the contest, another dataset (Reddit) was provided. It is formed of 232965 nodes, which is a significantly larger number with respect to the other datasets. As a result, the accuracy is naturally low for the given number of epochs.

1.3 Source Code

The graph convolutional network is composed of eight layers, which are reported in Figure 1 in their forward pass order.
Specifically, there are six unique building blocks, with Dropout and GraphSum being repeated twice:

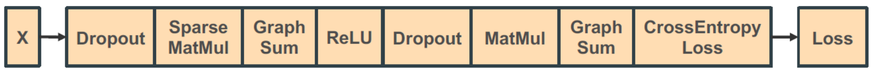


Figure 1: Graph Convolutional Network Layers

- Dropout: The dropout layer randomly sets input units to 0 with a frequency of P at each step during training time to prevent overfitting. Inputs that are not set to 0 are scaled up by 1/(1-P).
- Sparse Matmul: A sparse matrix multiplication layer.
- Graph Sum: A specialized sparse matrix multiplication for graphs.
- RELU: Rectified Linear Unit activation function. If the input is negative it will output 0.
- MatMul: A dense matrix multiplication layer.
- CrossEntropy Loss: Each predicted class probability is compared to the actual desired class and a loss is computed to penalize the probability based on how far it is with respect to the actual expected value. It is also called logarithmic loss.

The overall execution can be divided into three main phases: training, validation, and testing. In each phase, the forward pass is executed. This pass involves passing the graph data through the network's layers to generate the output. Moreover, the training phase also involves iteratively updating the weights of the network's layers using a technique called backpropagation (backward pass). During each epoch, both training and validation phases are conducted, while testing is performed at the end.

2 GPU Acceleration

2.1 Input and Truth

The first optimization in our code is the efficient management of input data and truth values. To restore the input data, we leveraged the GPU by saving the values in its memory and parallelized the restoration process using a kernel. Regarding the truth values, which can assume only three distinct forms, we have stored them in three separate GPU arrays. The selection of the appropriate array is based on the current epoch.

2.2 Layers

The Matmul, SparseMatmul, and GraphSum layers can be traced back to the following code for both forward and backward functions:

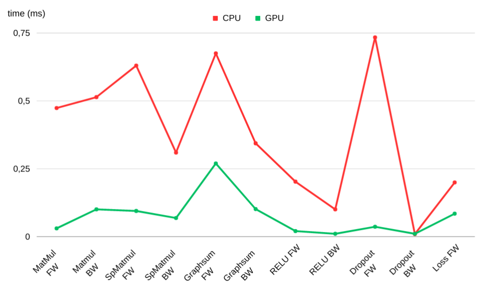
```
for(int i = 0; i < N; i++){
    for(int j = 0; j < M; j++){
        for(int k = 0; k < P; k++){
            v[i * M + j] = f();
        }
    }
}
```

In SparseMatmul and GraphSum the number of iterations of the for loop with k changes every time and is dependent on i. We can consider a variation of the code described above for these two functions using $P[i]$ instead of P in the for loop. In some optimization, we consider P as the maximum length of the for loop, that with this notation is $max(P[i])$ for every i.

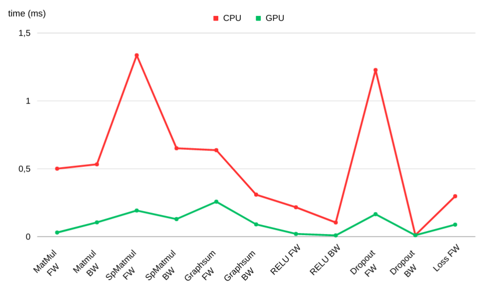
The naive GPU optimization consists of calling the kernel to parallelize variables i and j, the ones that do not cause race conditions, and executing the for loop with variable k. Starting from the naive solution we implemented different optimizations that are tailored and adjusted based on the specific requirements and dimensions of N, M, and P.

- **SQRT Decomposition**: The first optimization is based on the "Square Root Decomposition" algorithm. The for loop executes $\sqrt{P}$ operations instead of P and then the function *atomicAdd*() is used $\sqrt{P}$ times to sum the different values of $v[i \times M + j]$ calculated by each kernel call. This algorithm is efficient when the dimension of P is neither too small nor too big.
- **Index Sorting**: This optimization is applicable for SparseMatmul and GraphSum. The for loop is optimized by sorting the variables i in decreasing order based on the number of iterations of the for loop $P[i]$ that they have to execute. In this way the first blocks that are called will execute the longest for loops and the last blocks will execute the shortest for loops. With this optimization, the total time is lower because the last called blocks will finish their execution sooner. This is optimal when the average length of the for loop is very big, and it is not a constant value.
- **Sum Reduction**: The last optimization sums the values of $v[i \times M + j]$ using $log(P)$ kernel calls for every i and j, with an algorithm based on sum reduction. This is a very fast algorithm, especially when P has big values, but, when N and M are big, it exceeds the memory limitations and is not usable.

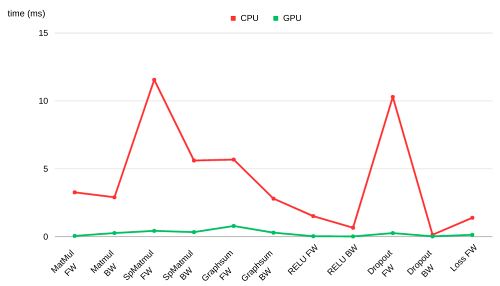
In some cases, we implemented different solutions: in MatMul forward we implemented a version that takes advantage of shared memory and is more efficient with big values of P; in SparseMatmul backward we implemented a different naive solution using the *atomicAdd*(). For CrossEntropyLoss, ReLu, and Dropout we implemented a simple solution parallelizing the for loop that they execute.



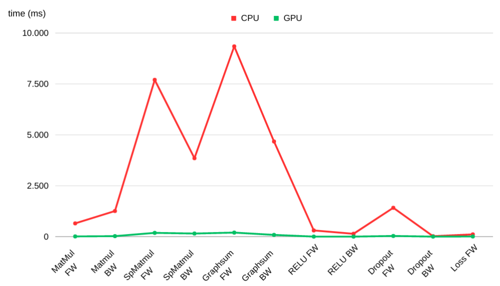
(a) Cora



(b) Citeseer



(c) Pubmed



(d) Reddit

Figure 2: Average Execution Time for each Layer Function

2.3 Conclusions

Below is a table displaying the average training time (per epoch) with and without GPU acceleration, and the related speedup. The mean values were computed based on 100 executions for Cora, Citeseer, and Pubmed, and 10 executions for Reddit, using Colab (GPU NVIDIA T4).

Dataset	CPU	GPU	Speedup
Cora	4.468 ms	1.088 ms	4.106
Citeseer	6.496 ms	1.878 ms	3.459
Pubmed	46.983 ms	2.189 ms	21.463
Reddit	29665.895 ms	734.155 ms	40.408

Table 2: Average Training Time Comparison

In our implementation, we maintained the original accuracy, with small oscillation due to the random seed initialization. For reference, the original accuracy was: 0.81900 for Cora; 0.77000 for Citeseer; 0.85400 for Pubmed; 0.21361 for Reddit.

In Figure 2 we plotted the average execution time for each layer function, both with and without acceleration. It is evident that as the dimension of the graph increases, we observe a greater speedup in performance. As a matter of fact, using the Nvprof profiler tool, we noticed that in Cora, which is the dataset with fewer nodes, the CUDA memcpy DtoH, memcpy HtoD, and memset take almost 10% of the total time of GPUs activities execution, while in Reddit it takes only 3%.

3 Model Optimization

3.1 Hyperparameters

For what concerns the model optimization phase of the project, we mainly focused on tuning the model hyperparameters to obtain a higher test accuracy, while taking into consideration the tradeoff with the execution time. Furthermore, we also tried to minimize the difference between the training and validation accuracy, where originally the former was smaller than the latter.
We examined four hyperparameters, which are:

- Hidden Dimension: A higher hidden dimension increases the model's capacity to fit the training data more closely, but at the same time increasing the hidden dimension leads to a larger model size and, as a consequence, to a higher execution time.
- Weight Decay: Weight decay is a form of L2 regularization that helps prevent overfitting by penalizing large parameter values. By decreasing the weight decay, the regularization effect is diminished. This allows the model to have more freedom in learning the training data.
- Epochs and Early Stopping: Early stopping is a technique used in machine learning to prevent overfitting and determine the optimal training epoch during the training process of a model. It consists in stopping the training process when the performance on the validation set starts to degrade. This potentially leads to better performance, but also to a longer training time.
- Learning Rate: A higher learning rate allows the model to update its parameters more aggressively during each iteration of the optimization algorithm. This can lead to faster convergence toward the optimal solution, but if the learning rate is too large, the model may update its parameters with excessively large steps, causing it to oscillate or diverge.

In Cora, Pubmed, and Citeseer, we diminished the weight decay (5e-4 $\rightarrow$ 1e-5). In Cora, we also increased the hidden dimension (16 $\rightarrow$ 32), while in Pubmed we increased the number of epochs (100 $\rightarrow$ 200) and enabled early stopping, considering the degradation with respect to the last 10 epochs. In Reddit, we diminished the weight decay (5e-4 $\rightarrow$ 0), increased the number of epochs (100 $\rightarrow$ 200), enabled early stopping (0 $\rightarrow$ 10), and increased the learning rate 0.01 $\rightarrow$ 0.02). The results of these changes can be seen in Table 3.

Dataset		Test Loss	Test Accuracy	Training Time
Cora	Not-optimized	1.08921	0.81900	1.088 ms
	Optimized	0.47895	0.87400	1.534 ms
Citeseer	Not-optimized	1.20771	0.77000	1.878 ms
	Optimized	0.78969	0.79100	2.463 ms
Pubmed	Not-optimized	0.60244	0.85600	2.189 ms
	Optimized	0.33759	0.88000	2.939 ms
Reddit	Not-optimized	3.20899	0.21361	734.155 ms
	Optimized	3.10091	0.23308	709.919 ms

Table 3: Model Optimization Results

3.2 Activation Function

We replace the RELU with the Swish Activation Function, but we did not obtain any improvement. The code for the two kernels is provided below.

```
__global__ void gpu_swish_forward (
    float *in_data, float *in_grad,
    float *mask, float *mask_grad,
    const bool training, const int idx_max)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= idx_max) return;
    in_data[i] = in_data[i] / (1 + exp(-in_data[i]));
}

__global__ void gpu_swish_backward (
    float *in_data, float *in_grad,
    float *mask, float *mask_grad,
    const int idx_max)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= idx_max) return;
    const float sig = 1 / (1 + exp(-in_data[i]));
    const float grad = sig * (1 + in_data[i] * (1 - sig));
    in_grad[i] = in_grad[i] * grad;
}
```