Optimization of Sparse Deep Neural Networks using GPU

Amruta Gokhale
EECS, IIT Bhilai
Raipur, India
amrutagokhale@iitbhilai.ac.in

Ankita Kumari
EECS, IIT Bhilai
Raipur, India
ankitakumari@iitbhilai.ac.in

Nidhi Sinchana SR
EECS, IIT Bhilai
Raipur, India
nidhisinchana@iitbhilai.ac.in

***Abstract*—Deep Neural Networks are an integral part of the field of Machine Learning. However, as data is continuously growing, the size of these neural networks is also increasing; thus also increasing memory dependence and hardware dependence, which is expensive. This introduces a need for finding a way to optimize calculations in this field. Sparse DNNs are a way to not only lessen the use of memory, but also keep the performance levels same. This paper aims to use the GPU to perform operations on matrices so that performance is optimized.**

I. Introduction

In Machine Learning, deep neural networks play a very important role. They are used in various fields such as image recognition, natural language processing, medical anomaly identification. [1] Since they have such a wide range of problems that they solve, the size of Deep Neural Networks is exceeding the capacity of the hardware to store them and train them at a fast pace. Thus arises the need to have a way to go about this calculation in such a way that the capacity and time bounds do not cross their available limits.

Many times data is stored in such a way that it becomes sparse which may make old existing solutions inefficient. Therefore, sparse DNNs were introduced to lower memory usage but keep the performance comparable to DNNs. In these sparse DNNs, data which is insignificant or not required is pruned away. Therefore the MIT graph challenge was introduced with the aim of optimizing sparse DNNs using GPUs.

The approach that we will be using will be using the W values in CSR Format. For the Y values, we will use two arrays to keep record of how many rows are non-zero and how many zeroes are there in each row. [2] This will help to reduce matrix multiplication with rows that are zero and also help with counting how many rows in the final matrix are non-zero. Along with that, pinned memory will also be used.

II. Approach

In our approach, the following steps were followed:

In the main function:

The input matrix Y is of dimension 60000×1024. We made two arrays for this matrix, i.e. a `nonZeroCount` and `emptyRows`. The array emptyRows stores the value 1 for rows which are empty.Since these rows contain only zeroes, they don't need to be multiplied. And the array nonZeroCount stores the number of non-zero values in a particular row.

For the layer matrices W, we used pinned memory. Whenever cudaMemCpy is called, first data is brought from paged memory to pinned memory and then to GPU memory. For saving time, we directly put values in pinned memory. However, since the size of pinned memory is small, we keep only 2 layer matrices at a time in it. First the older pinned memory is freed, and then the next W matrix is added to the pinned memory.

In the kernel function:

Shared memory is used to load both W and Y matrices. However, since shared memory size is small, we use *tiling* approach to load small parts of the matrices in tiles. The width of the tiles is taken to be 32. As the dimensions of Y are 60000×1024, the total number of tiles are 1875×32. The dimensions of W are 1024×1024, and therefore the number of tiles are 32×32. Each thread in the GPU calculates one element of the output. Each thread block is used for one tile. After the matrices are mutiplied, the bias is added and the *ReLu* function is applied. If the value is greater than 32, it is reduced to 32 and if the value is less than 0 it is increased to 0.

Once the output is obtained, instead of copying it from device to host and then back to device, it is directly copied from the GPU to GPU.

III. Evaluation Methodology

The hardware platform that was used for experiments is the GPU which is on Google Colab. The version of GPU that was used was Nvidia Tesla T4. It has a shared memory of 64KB for each SM and the maximum threadblock size is 1024.

Input data sets were be obtained from the website of MIT Graph challenge, for 1024 neurons.

We compared our implementation to the MATLAB serial version code that is given on the website of the challenge as well as the parallel implementation shown in the research paper *A GPU Implementation of the Sparse Deep Neural Network Graph Challenge* [3].

IV. Experimental Results

The challenge organizers provided results to compare our implementation to, and the parallel implementation also pro-

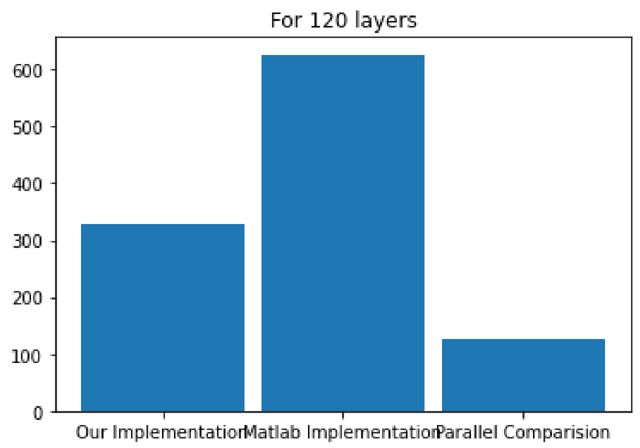

Fig. 1. Comparison for 120 layers

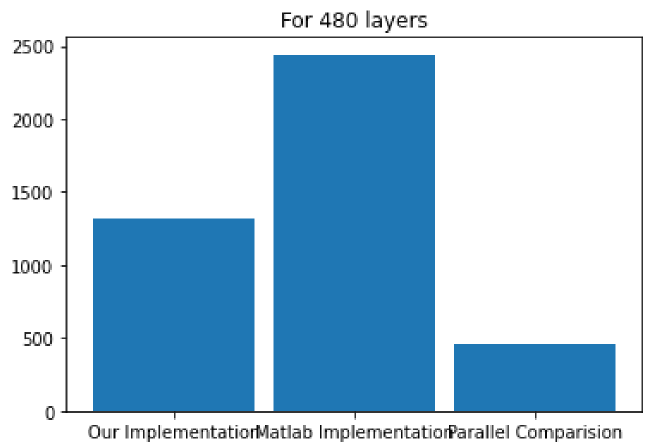

Fig. 2. Comparison for 480 layers

vided their results. We compared the results for 1024 layers.

IMPLEMENTATION	120 Layers	480 Layers
MATLAB	626 s	2440 s
Our Implementation	329 s	1320 s
Parallel Implementation	125 s	456 s

We can see that our implementation took half the time needed by MATLAB, however the parallel implementation was superior than ours. Note that they might have used a GPU superior than ours in the results that they provided and we used.

V. Related Works

Various works have been done on the sparse DNN challenge. The work that we compared our work to uses CSR representation of the input matrix for optimization, as it uses less memory and stays constant throughout all layers. Many approaches also used GRAPHBLAST for carrying out their computations. Some implementations used multiple GPUs for the optimization [4].

VI. Conclusion

In this work, we discussed why sparse matrices are needed because of the growing memory and time requirements. Also, how we can optimize multiplication of sparse matrices. We compared our optimizations to the original serial MATLAB code and also a parallel implementation. While our implementation took half the time of the matlab code, it was slower than the parallel implementation.

References

[1] J. Wang et al., ”Performance of Training Sparse Deep Neural Networks on GPUs,” 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1-5, doi: 10.1109/HPEC.2019.8916506.
[2] M. Bisson and M. Fatica, ”A GPU Implementation of the Sparse Deep Neural Network Graph Challenge,” 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1-8, doi: 10.1109/HPEC.2019.8916223.
[3] M. Bisson and M. Fatica, ”A GPU Implementation of the Sparse Deep Neural Network Graph Challenge,” 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1-8, doi: 10.1109/HPEC.2019.8916223.
[4] M. Hidayetoglu et al., ”At-Scale Sparse Deep Neural Network Inference with Efficient GPU Implementation,” 2020 arXiv, doi: 10.48550/ARXIV.2007.14152.