Accelerating Sparse Deep Neural Networks using GPUs and Parallel Programming.

Dakshraj Sadashiv Kashid
Dept. of EECS
IIT Bhilai
dakshrajsadashiv@iitbhilai.ac.in

Ashish Kumar
Dept. of EECS
IIT Bhilai
ashishkumar@iitbhilai.ac.in

Anshul Mandar
Dept. of EECS
IIT Bhilai
anshulmandar@iitbhilai.ac.in

***Abstract*—The Deep Neural Network has significantly improved the result of many machine learning problems. Due to its effective result, it has become a necessity to have a DNN model across various domains. But as the data is becoming vaster and vaster, the efficiency of DNN through normal computing is limited. Due to computation restrictions by standard machines, DNN cannot be extended to many layers which can effectively improve the result. . Therefore we propose to make use of parallel computing using GPU hardware to accelerate the implementation of Sparse DNN allowing more layers. Implemented parallel version of sparse matrix multiplication using partition and optimized SpMV kernel.**

I. Introduction

A. Problem Statement:

We have to enable large sparse deep neural network. The DNN is composed of many layers: l, and we are given weight matrices for each layer of the DNN W_l. We have to perform the ReLU equation for each layer $Y_{l+1} = h\{Y_l W_l + b_l\}$ to compute the layer-wise result [1]. Then we will compare the result of output layer with the given truth values to check the accuracy. The target is to achieve this result within minimum execution time along with comparable efficiency with the standard solution.

B. Motivation and Limitation to Existing Solution:

Today Deep Neural Networks (DNN) are used in many domains to solve complicated problems ranging from computer vision, natural language processing, speech recognition, and many more. It is always associated with massive data and it involves complex computation over a large data corpus. As large DNNs always tend to perform better but hardware restrictions and reasonable execution time impose a limitation on the structure of the DNN to be used. We can satisfy the DNN i.e compressing the weight matrices to a special format and with reduced memory usage and processing time it will give equivalent results..

C. Contribution:

Contribution: We will use GPU architecture to perform the parallel computation to evaluate the underlying calculation of the DNN. We will use a Hybrid format (ELL and COO) for the weight matrices and ELL format for Y matrices. This will reduce the execution time by reducing the memory load time and will efficiently utilize all the threads

II. Approach

A. Calculation.

Our aim is to calculate *ReLU* equation:$Y_{l+1} = h\{Y_l W_l + b_l\}$ for each layer. Now the bottleneck of this operation for inputs of large dimensions is sparse matrix to vector multiplication (*SpMV*), i.e. $Y_l * W_l$ part of the equation:

$$Y_{l+1} = h\{Y_l W_l + b_l\}$$

The weight matrices are sparse means they contain a large number of zeros in them, as these zeros will not have any effect on the above calculation it is redundant data. We will reduce these matrices using different storage methods as described in the following section. This compression greatly reduces the computational complexity of the problem.

As each thread will perform multiple operations it can result in race conditions. A condition where two or more threads try to update a single memory location concurrently, it may result in the loss of some updates. Thus we will synchronize this execution flow using atomic instructions of CUDA. Our primary objective for the synchronization of threads is, the maximum number of threads should be busy and not idle while handling the memory overload.

B. Storage format for the matrices.

The format in which matrices are stored are very important for the parallel execution of the program. There are various format in which the matrix can be stored such as CSR, COO, ELL and many more. CSR (*Compressed Sparse Row*) is general format for storing sparse matrices. It contains three arrays storing row-ptr, column-index and values. COO (Coordinate format) stores the row, column-indices and data in three arrays. ELL stores data in 2 matrices: ind and data, both of them have dimension: Number_of_rows*Maximum number_of_non-zero_in_any_row. [2]

The access from global memory in CUDA kernels slow down the execution due to high latency of operation. To counter this we have to store the processing data in kernel memory. Now threads in single warp collectively share shared_memory. It has high bandwidth of 1 TBPS and latency of 20-30 cycles as compared to that global memory with bandwidth of 200 GBPS and latency of 400-800 cycles

[5], but shared memory is limited in size. Therefore to reduce the delay caused by high latency of global memory, reduce shared memory usage and avoid redundant calculations we will use the following storage strategy:

1) We will store the weight matrices in CSC format, that is Compressed Sparse Columns. It provides minimum storage usage with high efficiency.
2) Now to calculate the inference, we need to send these matrices from host to device. Due to the large size of these matrices, it is infeasible to store all the matrices in a device in one go. As GPU memory is limited in size. But the number of host calls to transfer this data to a device can hinder the performance. We will minimize *cudaMemcpy* host calls.
3) The result of layer vector: Y_l will be loaded into the kernel only once. We do not need to repeatedly upload the updates because they can all take place on the device itself.

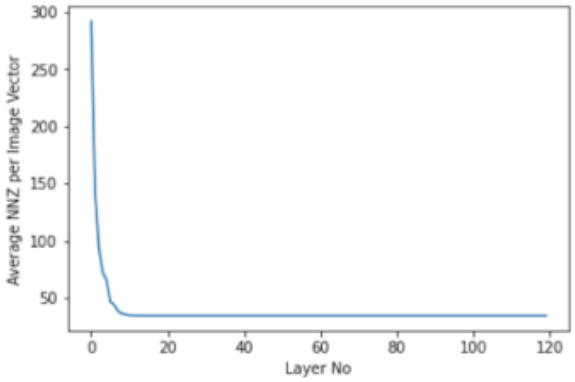

Fig. 1. The NNZ entries in feature vector over the iterations of weight layers.

C. Implementation

We implemented an optimized SpMV kernel. The kernel takes the weight matrix and feature vector as its parameter. Each thread will calculate the ReLU inference on a single row of the weight matrix.

1) For each layer, the inference is calculated for all the images while the weight layers reside in the device memory. This reduces the overhead to transfer the weight matrix data.
2) As it can be seen that many images get all zero entries in very few initial iterations.[Fig.1]. Thus keeping an *all_zero* flag to check whether all the entries of Y_l is zero. This omits extra iterations of inference.
3) As number of active neuron does not changes over the iteration, we will give coalesced memory access to W_l. Each thread will update one entry of Y_l and there are seven read/write operation on Y_l per thread, thus we will store it in shared memory to decrease latency.

The kernel is implemented for variable range of images, thus we provide inference calculation per image, rather than whole data set. A matrix-matrix multiplication can be more effective if the all feature vector can reside into the GPU memory.

III. Evaluation Methodology

A. Hardware platform

Using google colab for the experimental setup. The hardware configuration of the colab environment:

1) RAM: 12GB
2) GPU: NVIDIA Tesla K80
3) CPU: Intel(R) Xeon(R) CPU @ 2.00GHz

B. Input Data Set

We are using sparse MNIST input data, which is a corpus of 60,000 28x28 pixel images of handwritten numbers. [3] This data set contains images of handwritten numbers and we will use Deep Neural Network to predict the number and compare the truth categories given.

C. Comparison

Using the standard kernel for SpMV_CSR format and serial SpMV as baseline. The speed-up using the parallel programming can be compared with the seriel implementation [4] that of kernel optimization can be done using the performance of simple SpMV_CSR kernel.

IV. Experimental results

Execution Results for 1024 neurons:

Layer	*Execution Time(s)*	*Inference Time(s)*
120	64	26.796754
480	116	68.763494
1920	338	236.641190

The total execution time is affected by access and data transfer from google drive. It includes storing the external data into program memory. Whereas inference time is factored by kernel to host data transfer and inference calculation.

Now the comparative performance of inference vs total execution:

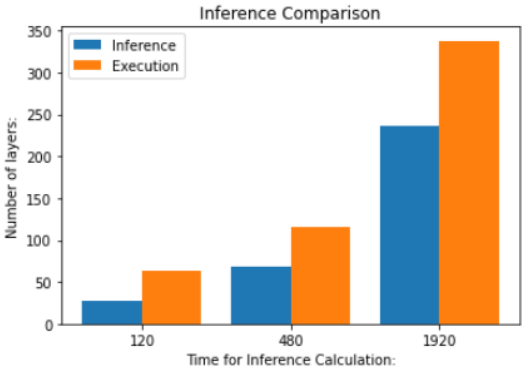

Fig. 2. Calculative execution overhead as compared to inference.

The execution overhead increases as the number of layers increases.

Now performance comparison between the Simple-CSR-based implementation and our Optimized SpMV inference kernel.:

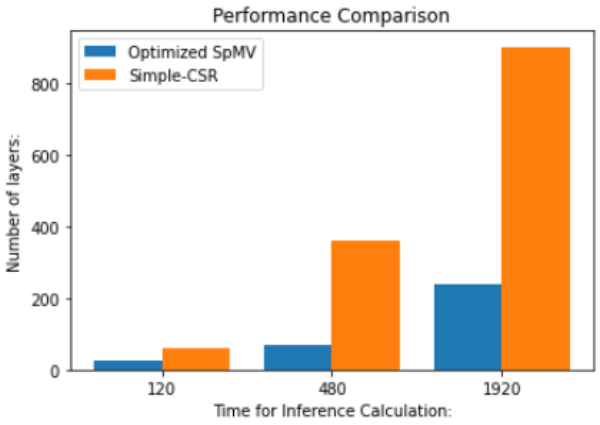

Fig. 3. Calculative execution overhead as compared to inference.

V. Related Work

The major portion of related work focuses on achieving fine-grained parallelism, memory bandwidth efficiency and compact storage formats. One such example is paper titled*"Efficient Sparse Matrix-Vector Multiplication on CUDA"* [5] focuses on usage of different storage format according to structure on matrices. It uses different storage format such as COO, ELL, DIA and CSR over varietal matrices to improve performance. But the problem is improper workload balance among threads leads to many idle threads.

Another front in this field is about workload distribution and low latency for GPU threads. They make use of CUDA block-thread architecture to make advantage of coalesced memory access and better cache performance.Such implementation can be seen in [6] and [7]. But such implementation only have constraints on supporting format and cannot give better performance in all cases of irregular matrices.

VI. Conclusion

We use the simple CSR_SpMV as baseline. From [Fig.1] we can infer that, the majority images become all_zero. Thus we use a flag to check that a feature vector is all_zero. By avoiding extra inference using the flag the speed is about 2.3x for 480 layers and 3.81x for 1920 layers.

As each row has 32/64/128 activated neurons. Thus the number of activated neuron over all the layers is constant. For the given input data set, there is no significant improvement upon partitioned based optimization. The number of non-zeroes entries(NNZ) in the feature vector also saturate over 32/64/128 non-zero. The kernel iterates over the NNZ entries of weight matrices so that the kernel calculation overhead doesn't change.

The inference kernel access the single entry of W_l only once. Storing weight matrices in shared memory has execution overhead upon single access. But every thread has irregular access of Y_l and include total of seven read/write operation. To increase memory access bandwidth Y_l is stored in shared memory, it provides 0.93x additional speed-up.

References

[1] https://graphchallenge.mit.edu/sites/default/files/documents/SparseDNN-GraphChallenge-2019-09-01.pdf
[2] https://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf
[3] https://graphchallenge.mit.edu/data-sets
[4] https://github.com/graphchallenge/GraphChallenge/blob/master/SparseDeepNeuralNetwork/matlab/runSparseDNNchallenge.m
[5] https://www.researchgate.net/publication/228680178_Efficient_Sparse_Matrix-Vector_Multiplication_on_CUDA
[6] https://ieeexplore.ieee.org/document/9664223
[7] https://lukeo.cs.illinois.edu/files/2015_GuGrOl_gpu.pdf