Data Parallel Large Sparse Deep Neural Network on GPU

Naw Safrin Sattar and Shaikh Arifuzzaman
Department of Computer Science
University of New Orleans
New Orleans, LA-70148, USA.
Email: {nsattar, smarifuz}@uno.edu

***Abstract*—Sparse Deep Neural Network (DNN) is an emerging research area since deploying deep neural networks with limited resources is very challenging. In this work, we provide a scalable solution to the Sparse DNN Challenge–a challenge posed by MIT/IEEE/Amazon GraphChallenge.org–by designing data parallelism on GPUs. We provide a solution based on Python TensorFlow as it is a widely used tool in different scientific applications for deep learning. We use the datasets provided by GraphChallenge, derived from the MNIST handwritten letters. We use the Synthetic DNNs from RadiX-Net with varying number of neurons and layers. We implement a data parallel implementation of Sparse DNN using TensorFlow on GPU. Our solution shows up to 4.7× speedup over the baseline serial MATLAB implementation given in GraphChallenge. In addition to that, our TensorFlow GPU implementation demonstrates a 3-fold speedup over our TensorFlow CPU implementation.**

***Keywords*-deep neural network; sparse data; parallel computing; GPU; TensorFlow**

I. Introduction

Nowadays, we have been observing a rapid growth of deep learning applications in business, scientific and technical domains [1]–[4]. Such applications are assumed to reign our lives in near future. For example, deep learning plays an important role in the field of speech recognition, visual object recognition, object detection, drug discovery, and genomics [5]. To address important problems in data-intensive computing [1], e.g., extracting complex patterns from massive volumes of data, semantic indexing, sentiment analysis, data tagging [2], fast information retrieval [3], [4], and simplifying discriminative tasks [6], deep learning techniques are being utilized heavily. However, deep learning models these days require a significant amount of memory and computing power which become a bottleneck in the conditions where such resources are limited [7]. Deep Neural Networks (DNNs) are often much harder to train than shallow neural networks. Larger neural networks often perform better because larger number of layers/features allow more non-linear boundaries. However, such larger networks are constrained by large memory requirements. Therefore, large-scale optimization [8]–[12] is needed to address such challenges [13], [14]. Sparse (pruned) neural networks deliver comparable performance with less amount of memory resources. In order to use machine learning features on mobile devices (facial recognition or voice assistants for instance), one needs small neural networks. Such on-the-go applications are another important motivation for using Sparse DNNs. With an increasing number of layers and neurons, the weight matrices can be made sparse to tackle the memory limitations. Pruning [15] results in better generalisation results, improved speed of processing the results and a reduced size as well. The need for sparse DNNs has inspired the Sparse Deep Neural Network Graph Challenge by MIT/IEEE/Amazon GraphChallenge community. The GraphChallenge seeks contribution from community of researchers to come up with solutions in the field of sparse data and graph analytics [16]–[18] by collecting data from different scientific domains [19]. Sparse Deep Neural Network Graph Challenge performs neural network inference on a variety of sparse deep neural networks.

In this paper, we present a solution to the Sparse DNN Graph Challenge using Python TensorFlow. We achieve up to 4.7-fold speedup using GPUs over the serial MATLAB implementation provided in GraphChallenge by implementing Sparse DNN in data parallel mode. We also compare different strategies of Distributed Training API of TensorFlow and demonstrate their performance empirically.

The rest of this paper is organized as follows. We describe the related works in Section III. Background of Deep Neural Networks and the parallelization techniques are discussed in Section II. In Section IV, we describe our methods to implement a TensorFlow based data parallel sparse DNN using different strategies of distributed training of TensorFlow in CPU as well as GPU. Our datasets and experimentation environment are described in Section V. A detailed analysis of the results is presented in Section VI . Finally, Section VII summarizes the findings and concludes the paper with a discussion of future possibilities.

II. Preliminaries

In this section, we discuss the basic concepts of DNN and the scope for parallelization of our serial baseline methodology.

A. How DNNs work

A sample DNN is illustrated in Fig. 1. The primary mathematical operation performed by a DNN is the inference, or

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00170

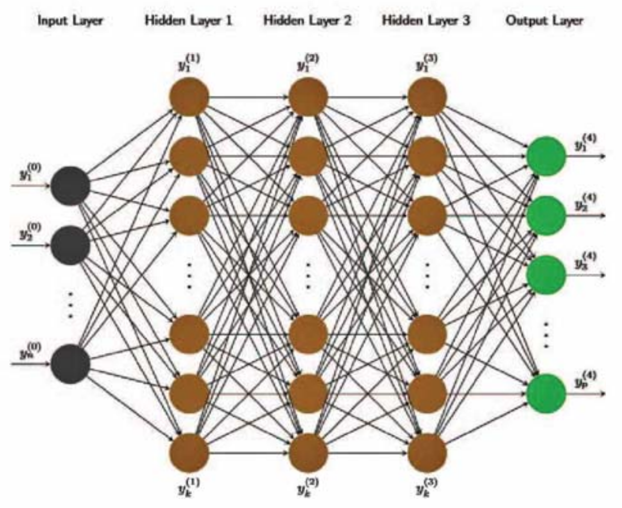


Figure 1: Four layer ($L = 4$) deep neural network architecture. The input features $y^{(0)}$ are passed through a series of network layers $W^{(l=0,1,2,3)}$ with bias terms $b^{(l=0,1,2,3)}$, that produce scores for categories $y^{(L=4)}$

forward propagation step. Inference is executed repeatedly during training to determine both the weight matrix $W^{(l)}$ and the bias vectors $b^{(l)}$ of the DNN. The inference computation is given by Equation 1.

$$y^{(l+1)} = h(y^{(l)}W^{(l)} + b^{(l)}) \tag{1}$$

where, $h()$ is a nonlinear function applied to each element of the vector. A commonly used function is the rectified linear unit (ReLU) given by Equation 2. For the Sparse DNN challenge, $h()$ also has an upper limit set to 32. ReLU function sets negative values to 0 and values greater than 32 to 32.

$$h(y) = \begin{cases} y & \text{if } 0 < y < 32 \\ 0 & \text{if } y < 0 \\ 32 & \text{if } y > 32 \end{cases} \tag{2}$$

$W(i, j) > 0$ implies a connection between neuron i and neuron j, using the standard graph community convention. Dimension of $W^{(l)}$ matrix is $k \times k$ where, k denotes the number of neurons.

The quantities $y^{(l)}$ are row vectors. Each row is a feature vector. In Sparse DNN Challenge, according to standard graph community terminology, left matrix multiply is used to progress through the network. Transposing all matrices and multiplying on the right can be used according to Standard AI definitions. The quantity $b^{(l)}$ is a bias row vector applied to each input.

When training a DNN, or performing inference on many different inputs, it is usually necessary to compute multiple $y^{(l)}$ vectors at once in a batch that can be denoted as the matrix $Y^{(l)}$. The dimension of matrix $Y^{(l)}$ is $n \times p$ where, n = number of inputs, p = number of features. In matrix form, the inference step follows Equation 3.

$$Y^{(l+1)} = h(Y^{(l)}W^{(l)} + B^{(l)}) \tag{3}$$

Here, $B^{(l)}$ is a replication of $b^{(l)}$ along columns given by Equation 4.

$$B^{(l)} = b^{(l)}|Y^{(l)}one|_0 \tag{4}$$

Here, *one* is a column array of 1's, and $|\ |_0$ is the zero norm. The overall steps to calculate inference is given in Algorithm 1.

Algorithm 1: Serial Inference Calculation in MATLAB

```
Data: $W$ is the Weight Matrix,
      $bias$ is the Bias Vector,
      $Y0$ is the Input, MNIST Image
Result: $Y$
1  $Y_{max} \leftarrow 32$
2  $Y \leftarrow Y0$
3  $nlayers \leftarrow length(W)$
4  for $layer$ until $nlayers$ do
5      $Z \leftarrow Y \times W\{layer\}$
6      $b \leftarrow bias\{layer\}$
7      $Y \leftarrow Z+$bsxfun(@times, double
        (logical($Z$)), $bias\{layer\}$)
                // bsxfun(@$x$, $a$, $b$) applies $x$
        operation on arrays $a$ and $b$
8      if $Y < 0$ then
9          $Y \leftarrow 0$
10     end
11     else if $Y > Y_{max}$ then
12         $Y \leftarrow Y_{max}$
13     end
14 end
```

B. Scopes for Parallelization

We aim at scaling up our sparse DNN based computing task through parallel implementation. To parallelize the DNN, we consider the following three possibilities.

1) Data Parallelism : In data parallelism [20], we split inputs across all devices. The weight matrices need to be replicated on every processor. This method might not be the most space efficient but the design of such method is comparatively less complex.

2) Model Parallelism: This paradigm requires splitting up the layers and running in a pipeline parallel mode. It requires communicating results after each group of layers. It saves memory but the design is more complex. Achieving balanced loads among multiple GPUs is very challenging. The complexity of different DNN layers varies, introducing significant efforts for programmers to partition model layers to GPUs in a balanced way. In order to achieve pipeline parallelism [21], understanding the mapping pattern of the communication is vital. Pipeline parallelism, itself, is a separate field of research. Another shortcoming of this method is to determine where the neural network is being split. The neural network requirements to process data also needs to be increased significantly. The weight staleness issue [22] is another concern. Since gradients are computed with stale weights, training instability and accuracy loss are persistent.

3) Hybrid Parallelism : This is a combination of data and model parallelism. Benefits from both of them can be combined in a hybrid [23]–[25] implementation.

In this paper, we use data parallelism for our parallel implementation in an attempt to scale up our sparse DNN based task. We use TensorFlow Distributed Training to implement data parallelization for Sparse DNN in our experimentation. We have left model and hybrid parallelism as our future work.

III. Related Work

There exist several work in the current literature that focus on accelerating deep neural networks [26]–[32]. As provided by *GraphChallenge*, a serial algorithm for Sparse DNN is given in [27]. The MATLAB serial reference of the inference calculation is given in Algorithm 1. They present the serial timing measurements of the MATLAB code to be used as a benchmark. They also provide parallel implementation of the Sparse DNN benchmark. They parallelize the code by splitting feature vectors, then develop and test it on the MIT SuperCloud TX-Green supercomputer, Intel KNL processors with 192 GB RAM using pMatlab.

Another solution to the Sparse DNN Challenge has been provided by Davis et al. [28] using GraphBLAS [33]. The sequential performance of the GraphBLAS solution is $3\times$ to $5\times$ faster than the MATLAB reference implementation. OpenMP parallelism gives an additional $10\times$ to $15\times$ speedup on a $20-$core Intel processor, $17\times$ on an IBM Power8 system, and $20\times$ on a Power9 system, for the largest problems. The performance metric, *Rate* measures the throughput of the implementation as the ratio of the number of inputs times the number of connections in the DNN divided by the execution time [27]. There is an inconsistency in using the *Rate* formula in serial MATLAB code and their GraphBLAS implementation. The discrepancy between the computed rate for MATLAB code in [28] and [27] can be misleading to the readers. This variation happens as the number of inputs is not included in the numerator for *Rate* calculation in [28] for MATLAB code. Another Sparse DNN Challenge solution given in [26] shows a GPU implementation of the GraphBLAS standard. Their implementation shows a $1.94\times$ speedup over the "SuiteSparse" CPU implementation of GraphBLAS.

Apart from the Sparse DNN challenge, the authors in [29] have proposed redundancy reduction schemes, including the software/hardware co-designs of the structured sparse neural network, an enhanced LRA algorithms and the ternary quantized gradients training for the distributed DNN. In [30], the authors propose a multiscale kernels approach to extract optimal criteria for saliency detection where they suppress nonsalient regions in the sparse labeling. The authors find an interesting way to connect nodes in a sparsely connected network [31]. Their sparse evolutionary training of artificial neural networks algorithm evolves an initial sparse topology (Erdős–Rényi random graph) of two consecutive layers of neurons into a scale-free topology during learning. Fully-connected layers of artificial neural networks are replaced with sparse ones before training and the parameters are reduced quadratically with no loss in accuracy. In [32], the authors develop a Structured Sparsity Learning (SSL) method to regularize the structures (i.e., filters, channels, filter shapes, and layer depth) of DNNs, that can learn a compact structure from a bigger DNN to reduce computation cost.

Our work is different from the work of [26], [28] as we focus on an efficient data parallel implementation using Python. Several deep learning frameworks, i.e., Tensorflow [34], PyTorch [35], are written in Python. Many scientific applications use these widely used deep learning frameworks. GraphBLAS focuses mainly on graph algorithms and is used by a specific community of graph researchers. Our work is generic in nature–it will help the end-users of different domains to apply these techniques with ease and achieve high performance capability.

IV. Methodology

We use Python TensorFlow to solve the Sparse DNN problem. We implement our own layers by extending the **tf.keras.Layer** class and implementing: $*$*__init__* , where we have done all input-independent initialization build. We know the shapes of the input tensors and can do the rest of the initialization call, doing the forward computation. For the activation function, we use the basic *matmul* function to implement our own *ReLU* function instead of the matrix multiplication given in Algorithm 1. A pseudocode for initialization of our custom layer is given in 2. We use this class *myClass* to build our model where the outer layer tracks the weights of the inner layer. For brevity, detailed code for model training is not described in the paper.

Initially, we start with a CPU implementation disabling CUDA. We divide the 60,000 samples into 60 chunks of 1000 samples/chunk. Later on, we activate a single GPU to check its performance. We use the Central Storage Strategy from tf.distribute.Strategy API where all variables and operations will be placed on the GPU if a single GPU is available. This strategy is experimental and may change in future.

With multiple GPUs, we can use the extra computing power effectively by increasing the batch size. Using the

Algorithm 2: A Pseudocode for creating custom layer to build our own model in TensorFlow

```
Data: $W$ is the Weight Matrix,
      $bias$ is the Bias Vector,
      $Y0$ is the Input, MNIST Image
$Y_{max} \leftarrow 32$
/* The custom layer class
   encapsulating both the layer's
   "weights" and a transformation
   from inputs to outputs           */
class myClass(layers.Layer)
__init__()          // Input dimension and
  general initialization
build()        // Initialization of layer
  weights and bias
call()        // The layer's forward pass
__init__()
{
  super.__init__()
  self.unit $\leftarrow$ unit
}
build()
{
  self.w$\leftarrow$ $W_0$
  self.b$\leftarrow$ $bias_0$
}
call()
{
  $Y \leftarrow$ tf.matmul(inputs, self.w) + self.b
  if $Y < 0$ then
    $Y \leftarrow 0$
  end
  else if $Y > Y_{max}$ then
    $Y \leftarrow Y_{max}$
  end
  return $Y$
}
```

largest batch size that fits the GPU memory utilizes the resources efficiently. But we have only 2 GPUs, limited by our system. So, we have used 60 chunks, batch size of 1000 throughout our experimentation.

For distributed training, we use tf.distribute.Strategy API. This is a TensorFlow API to distribute training across multiple GPUs, multiple machines or TPUs. We use the GPU functionality only. There are 6 different types of strategies available. But we choose to work with **tf.distribute.MirroredStrategy** because it is fully supported by Keras API and other strategies are still in experimental support phase. Mirrored Strategy supports synchronous distributed training on multiple GPUs on one machine. It creates one replica of the model per GPU device. Each variable in the model is mirrored across all the replicas. Together, these variables form a single conceptual variable called MirroredVariable. These variables are kept in synchronization with each other by applying identical updates. *MirroredStrategy* takes care of replicating the model's training on the available GPUs, aggregating gradients, and more. Efficient all-reduce algorithms are used to communicate the variable updates across the devices. All-reduce aggregates tensors across all the devices by adding them up and makes them available on each device. It's a fused algorithm that is very efficient and can reduce the overhead of synchronization significantly. There are many all-reduce algorithms and implementations available, depending on the type of communication available between devices. By default, it uses NVIDIA NCCL as the all-reduce implementation. The other communication schemes are Hierarchical Copy All Reduce and Reduction To One Device. We try different communication schemes to achieve the best result.

Table I: Memory Requirement for various DNN size

Neuron per Layer	Layer	Total Neuron	Memory Required (GB)	Experiment Done
1,024	120	122,880	0.117	Yes
	480	491,520	0.469	Yes
	1,920	1,966,080	1.875	Yes
4,096	120	491,520	0.469	Yes
	480	1,966,080	1.875	Yes
	1,920	7,864,320	7.5	Only CPU, Exceed Local Memory for GPU
16,384	120	1,966,080	1.875	Yes
	480	7,864,320	7.5	Only CPU, Exceed Local Memory for GPU
	1,920	31,457,280	30	No
65,536	120	7,864,320	7.5	No
	480	31,457,280	30	No
	1,920	125,829,120	120	No

Later on, we experiment with the strategy *tf.distribute.experimental.MultiWorkerMirroredStrategy* to perfrom a comparative analysis of different strategies in multiple GPUs. Similar to MirroredStrategy, it creates copies of all variables in the model on each device across all workers. Two different implementations are available in the experimental support: *CollectiveCommunication.NCCL* and *CollectiveCommunication.Ring* for collective operations. We use both and select the best one depending upon the number and kind of GPUs available in our system, and the network interconnect in the cluster.

We could not experiment with the 65536-neuron case as its storage requirement significantly exceeds the available memory of our system. Table I shows the memory requirement for working with each of the different sizes of DNNs. Scaling to a large dataset require a different approach, most certainly exploiting model parallelism or combination of model and data both, a hybrid strategy.

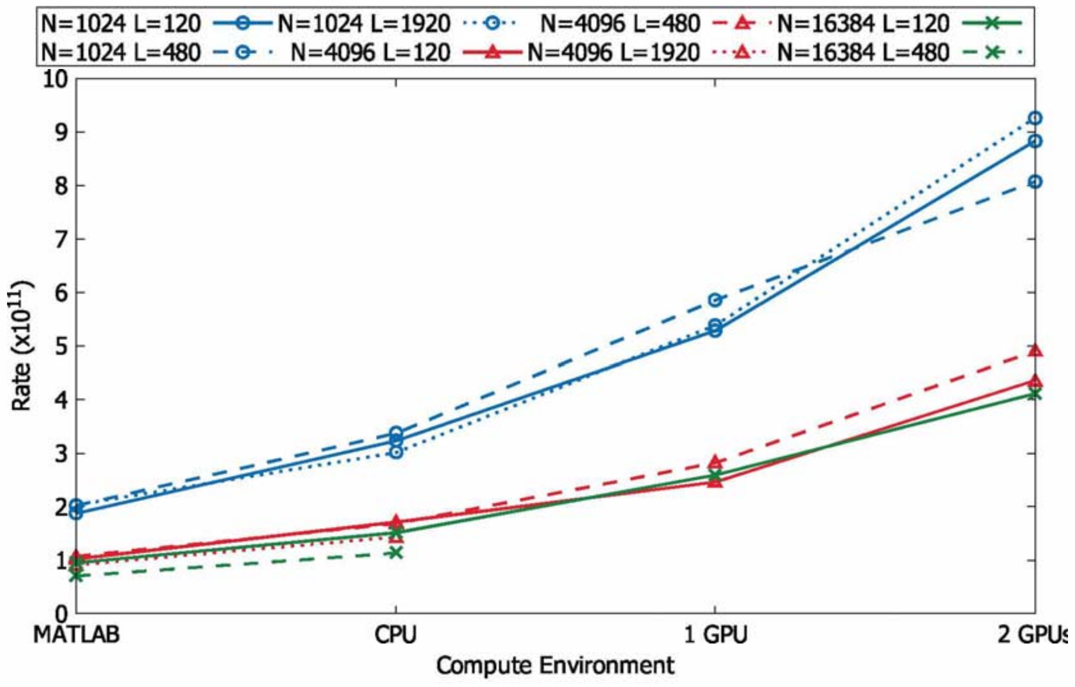


Figure 2: Change of Rate with increasing computing power compared to the MATLAB baseline for different size of DNNs

V. Experimental Setup

A. Environment

We used Louisiana Optical Network Infrastructure (LONI) to perform all the experiments. QB2 [36], a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz, 20 cores per node. A single node has two 960 NVIDIA Tesla K20x GPUs with 128GB memory, 500 GB HDD, that has been used for the experiments performed using GPUs. However, the available local memory for each GPU is limited to 5.566GB. QB2 has RedHat Enterprise Linux 6 Operating System, 56 Gb/sec (FDR) InfiniBand 2:1 oversubscribed mesh, 1 Gb/sec Ethernet management network, 10 Gb/sec and 40 Gb/sec external connectivity. NVIDIA Driver 396.51 has been used with CUDA 10.0 and the TensorFlow version is 1.14.

The serial MATLAB code has been run on an Intel Core i7-4770 CPU @ 3.4GHz×8 processor and 16 GB RAM machine with MATLAB version R2015a.

B. Dataset

Table II: MNIST Input Resizing

Neurons	Size	Input Dimension
1024	176 MB	32×32
4096	800 MB	64×64
16384	3.6 GB	128×128

Sparse DNN Challenge requires input data or feature vectors Y_0. MNIST (Modified National Institute of Standards and Technology) is a large database of handwritten digits that is widely used for training and testing DNN image processing systems. MNIST consists of $60,000$ 28×28 pixel images. Truth categories for MNIST are included for performing inference using DNN with specific numbers of layers. The Sparse DNN Graph Challenge uses interpolated sparse versions of this entire corpus as input.

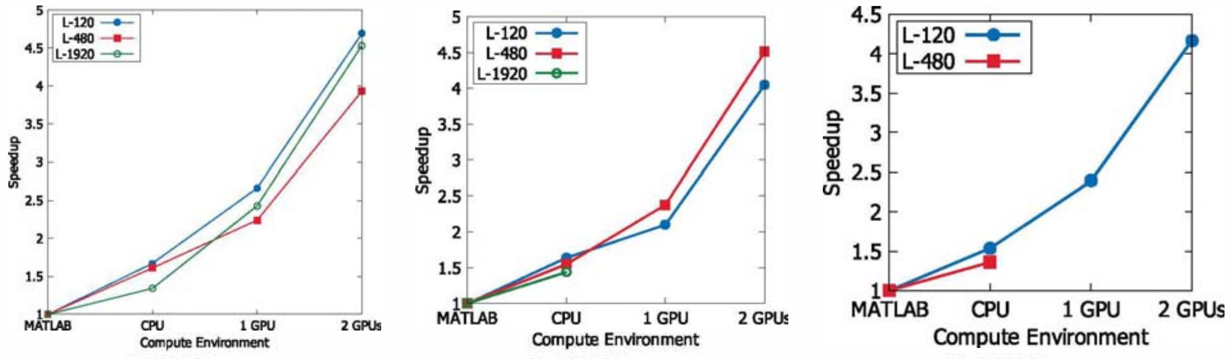
Sparse DNNs in MNIST corpus are resized to produce neural networks of varying dimensions shown in Table II. Each 28×28 pixel image is resized to 32×32 (1024 neurons), 64×64 (4096 neurons), 128×128 (16384 neurons), and 256×256 (65536 neurons). The resized images are thresholded so that all values are either 0 or 1. The images are flattened into a single row to form a feature vector. The non-zero values are written as triples to a .tsv file where each row corresponds to a different image, each column is the nonzero pixel location and the value is 1. We also use Synthetic DNNs created using RadiX-Net [37] with varying number of neurons and layers.

VI. Result

In this section, we describe the performance of our data parallel implementation of the Sparse DNN Challenge. While running the serial MATLAB implementation, we faced some difficulties because of the version compatibility issues of MATLAB. We needed to change some functions

Table III: Computational Result on Different Sparse DNNs using MATLAB on CPU; Python TensonFlow on both CPUs and GPUs, Speedup calculated w.r.t. Baseline MATLAB Serial Implementation

Neurons per Layer	Layer	Connection (Edges) $\times 10^6$	MATLAB		Python TensorFlow								
					CPU			1 GPU			2 GPUs		
			Time (sec)	Rate $\times 10^9$ /sec	Time (sec)	Rate $\times 10^9$ /sec	Speedup	Time (sec)	Rate $\times 10^9$ /sec	Speedup	Time (sec)	Rate $\times 10^9$ /sec	Speedup
1,024	120	3.93	126.3	1.87	75.603	3.23	1.67	47.5	5.28	2.66	26.914	8.83	4.69
	480	15.73	466.7	2.02	289.296	3.37	1.61	208.1	5.85	2.24	118.754	8.07	3.93
	1,920	62.91	1,862.01	2.03	1,391.672	3.01	1.34	765.74	5.38	2.43	410.986	9.26	4.53
4,096	120	15.73	921.2	1.02	562.649	1.71	1.64	437.7	2.46	2.10	227.528	4.36	4.05
	480	62.91	3,540.7	1.07	2,290.483	1.68	1.55	1,489.4	2.82	2.38	785.046	4.90	4.51
	1,920	251.66	16,579	0.91	11,475.7	1.43	1.44	Local Memory of GPU exceeded					
16,384	120	62.91	3,990.3	0.95	2,583.25	1.51	1.54	1,667.17	2.59	2.39	957.5571	4.11	4.17
	480	251.66	21,626	0.70	15,932.36	1.13	1.36	Local Memory of GPU exceeded					



(a) 1024 Neurons (b) 4096 Neurons (c) 16384 Neurons

Figure 3: Speedup of DNNs in Python TensorFlow CPUs and GPUs over the baseline MATLAB Implementation [L denotes number of Layers]

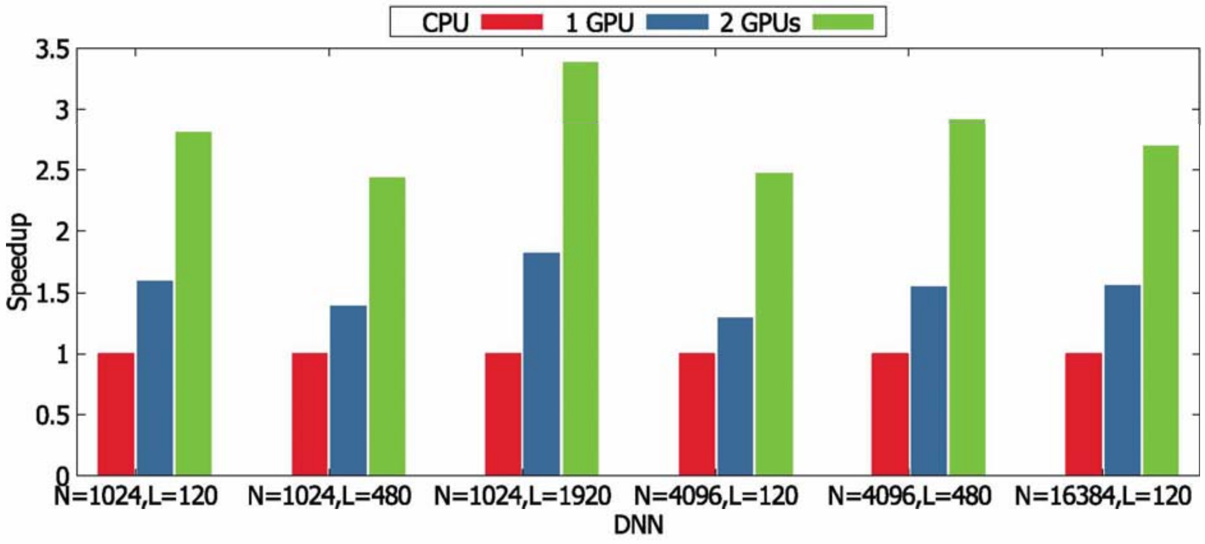


Figure 4: Speedup of DNNs in GPUs over the CPUs using Python TensorFlow [N-Neuron, L-Layer]

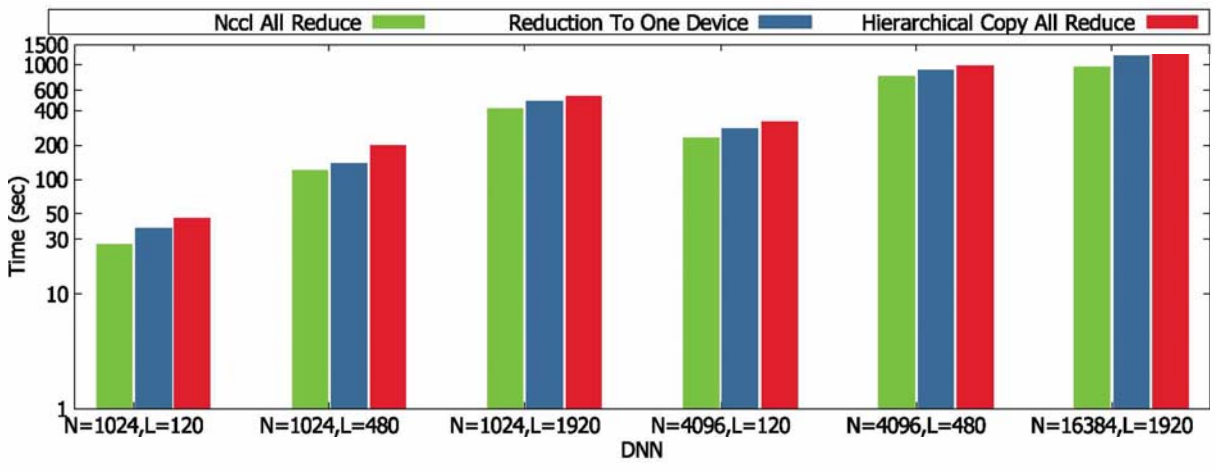


Figure 5: Change in Runtime for Different Cross Device Communication Patterns for Distributed Mirrored Strategy [N-Neuron, L-Layer]

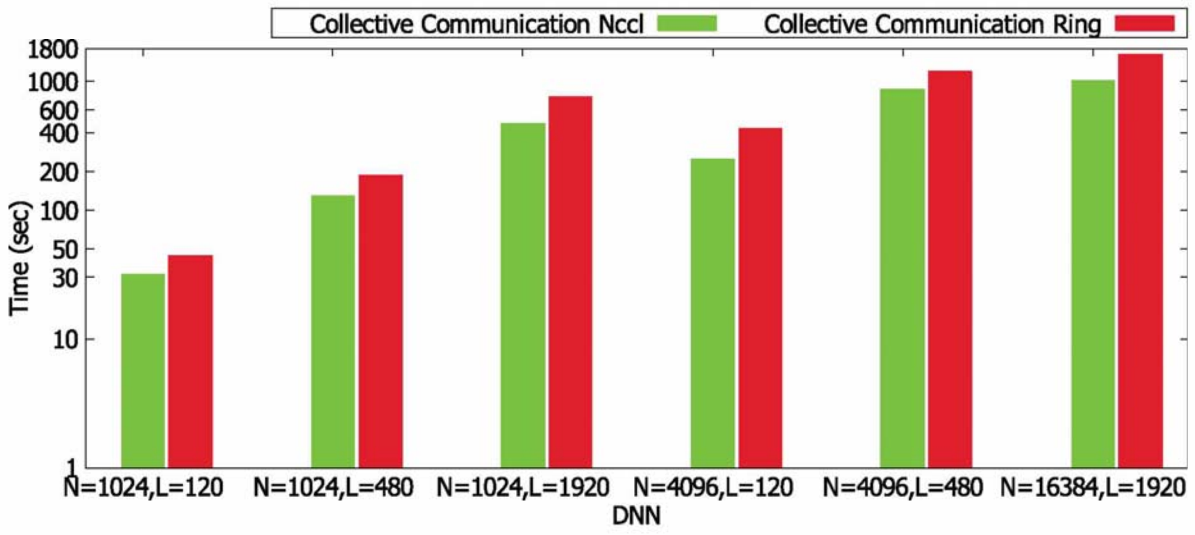


Figure 6: Change in Runtime for Different Cross Device Communication Patterns for Distributed Multi Worker Mirrored Strategy [N-Neuron, L-Layer]

to run the baseline serial MATLAB code. We have summarized the experimentations performed on different Sparse DNNs in Table III. We compute speedup for our CPU and GPU implementations keeping the MATLAB sequential code as baseline. We get upto 4.7× speedup in terms of computational time over the baseline sequential MATLAB implementation using 2 GPUs. The inference rate changes with increasing computation power for various DNN sizes shown in Figure 2. Our change of inference rate is similar in nature as given by J. Kepner using pMatlab [27]. The runtime speedup over the baseline MATLAB for different DNNs is depicted in Figure 3. For different numbers of neurons, how the speedup varies with changing layers is portrayed. In case of smaller number of layers (L=120), all of the DNNs show speedup in between 4 to 5.

Table IV: Performance Comparison of Different Strategies for Distributed Training in TensorFlow

Neurons per Layer	**Layer**	**Mirrored**	**Speedup**	**Multi Worker Mirrored**	**Speedup**
1,024	120	26.91	4.69	31.81	3.97
	480	118.75	3.93	127.22	3.67
	1,920	410.99	4.53	470.06	3.96
4,096	120	227.53	4.05	249.81	3.69
	480	785.05	4.51	867.33	4.08
16,384	120	957.56	4.17	1,015.23	3.93

Besides, our parallel implementation using GPUs, our CPU sequential implementation is $1.34\times - 1.67\times$ faster than the MATLAB sequential implementation. The speedup of our GPU implementation compared to our CPU implementation is shown in Figure 4. We get up to $3.38\times$ speedup using 2 GPUs over the CPU implementation. The performance is proportionally scalable with the available number of GPUs. We can use only 2 GPUs available in our system. The scalability can be increased by using more GPUs.

We also experiment with different strategies of TensorFlow Distributed Training. Mirrored Strategy works better than the Multi Worker Mirrored Strategy as shown in Table IV. The reason might be that Multi Worker Mirrored Strategy has Experimental Support whereas Mirrored Strategy is fully supported in TensorFlow. Mirrored Strategy as well as the Multi Worker Mirrored Strategy support different cross device communication methods. We experiment with each communication schemes and find out the best one. In case of Mirrored Strategy, the default scheme Nccl All Reduce outperforms the other two and takes less time shown in Figure 5. While experimenting with Multi Worker Mirrored Strategy, we find that Nvidia's NCCL works faster compared to the ring-based collectives using gRPC as the communication layer depicted in Figure 6.

VII. Conclusion and Future Work

Deep Neural Networks have become one of the prominent research topics in recent times to support modern Artificial Intelligence activities in current world. We face scalability difficulties while working with deeper neural networks falling under the domain of Big Data. The Sparse DNN Challenge given by MIT/IEEE/AmazonGraphChallenge.org focuses on developing new solutions to help the community solving the problems in the domain of graph analytics, machine learning, sparse dataset, big data, high performance computing, and visual analytics. In our work, we provide a solution to the Sparse DNN Challenge using Python TensorFlow. We have achieved $4.7\times$ speedup in data parallel mode in GPU over the baseline sequential MATLAB implementation. We have compared different strategies of TensorFlow Distributed Training and showed their performance. Our work will help the scientific community who use TensorFlow for deep learning in their application to achieve high performance. We get $2.9\times$ speedup using only 2 GPUs over our CPU implementation. We will perform experiments on large number of GPUs in other systems to show the scalability of our solution in our future work. In future, we plan to compare our performance with GraphBLAS implementation and other approaches in the same system used by those work. We will also work towards model parallelism and hybrid parallel implementations for Sparse DNNs.

Acknowledgment

This work has been partially supported by Louisiana Board of Regents RCS Grant LEQSF (2017-20)-RDA-25 and University of New Orleans ORSP SCORE award 2019. We also thank the anonymous reviewers for the helpful comments and suggestions to improve this paper.

References

[1] B. Jan, H. Farman, M. Khan, M. Imran, I. U. Islam, A. Ahmad, S. Ali, and G. Jeon, "Deep learning in big data analytics: a comparative study," *Computers & Electrical Engineering*, vol. 75, pp. 275–287, 2019.

[2] N. S. Sattar, S. Arifuzzaman, M. F. Zibran, and M. M. Sakib, "Detecting web spam in webgraphs with predictive model analysis," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019.

[3] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.

[4] H. Li and Z. Lu, "Deep learning for information retrieval," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, 2016, pp. 1203–1206.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 1, 2015.

[7] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi, "Sparse deep neural network exact solutions," *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep 2018. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2018.8547742

[8] N. S. Sattar, "Scalable community detection using distributed louvain algorithm," 2019.

[9] S. Arifuzzaman, M. Khan, and M. Marathe, "Fast parallel algorithms for counting and listing triangles in big graphs," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 1, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3365676

[10] S. Arifuzzaman and M. Khan, "Fast parallel conversion of edge list to adjacency list for large-scale graphs," in *Proceedings of the 23rd High Performance Computing Symposium (HPC 2015), Alexandria, VA, USA*, April 2015, pp. 17–24.

[11] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu, "Training deeper models by gpu memory optimization on tensorflow," in *Proc. of ML Systems Workshop in NIPS*, 2017.

[12] N. S. Sattar, T. Aqila, and R. Shahriyar, "Towards concurrent data structure development with relaxed synchronization," in *2016 9th International Conference on Electrical and Computer Engineering (ICECE)*. IEEE, 2016, pp. 267–270.

[13] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[14] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[16] N. S. Sattar and S. Arifuzzaman, "Overcoming mpi communication overhead for distributed community detection," in *Workshop on Software Challenges to Exascale Computing*. Springer, 2018, pp. 77–90.

[17] ——, "Parallelizing louvain algorithm: distributed memory challenges," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC)*. IEEE, 2018, pp. 695–701.

[18] S. Arifuzzaman, M. Khan, and M. V. Marathe, "PATRIC: a parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013), San Francisco, CA, USA*, October 2013, pp. 529–538.

[19] "Motivation." [Online]. Available: https://graphchallenge.mit.edu/

[20] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *arXiv preprint arXiv:1811.03600*, 2018.

[21] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.

[22] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and robust parallel dnn training through model parallelism on multi-gpu platform," *arXiv preprint arXiv:1809.02839*, 2018.

[23] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[24] A. A. Awan, A. Jain, Q. Anthony, H. Subramoni *et al.*, "Hypar-flow: Exploiting mpi and keras for scalable hybrid-parallel dnn training using tensorflow," *arXiv preprint arXiv:1911.05146*, 2019.

[25] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "Lbann: Livermore big artificial neural network hpc toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 1–6.

[26] X. Wang, Z. Lin, C. Yang, and J. D. Owens, "Accelerating dnn inference with graphblas and the gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[27] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," *arXiv preprint arXiv:1909.05631*, 2019.

[28] T. A. Davis, M. Aznaveh, and S. Kolodziej, "Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse: Graphblas," in *2019 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[29] B. Li, W. Wen, J. Mao, S. Li, Y. Chen, and H. H. Li, "Running sparse and low-precision neural network: When algorithm meets hardware," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 534–539.

[30] K. Yan, X. Wang, J. Kim, and D. Feng, "A new aggregation of dnn sparse and dense labeling for saliency detection," *IEEE Transactions on Cybernetics*, 2020.

[31] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature communications*, vol. 9, no. 1, pp. 1–12, 2018.

[32] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[33] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The graphblas c api specification," *GraphBLAS. org, Tech. Rep.*, 2017.

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[36] "Documentation — user guides — qb2," http://www.hpc.lsu.edu/docs/guides.php?system=QB2.

[37] R. A. Robinett and J. Kepner, "Radix-net: Structured sparse matrices for deep neural networks," *arXiv preprint arXiv:1905.00416*, 2019.