Multithreaded Layer-wise Training of Sparse Deep Neural Networks using Compressed Sparse Column

Mohammad Hasanzadeh Mofrad, Rami Melhem
University of Pittsburgh
Pittsburgh, USA
{moh18, melhem}@pitt.edu

Yousuf Ahmad and Mohammad Hammoud
Carnegie Mellon University in Qatar
Doha, Qatar
{myahmad, mhhamoud}@cmu.edu

***Abstract*—Training a sparse Deep Neural Network (DNN) is inherently less memory-intensive and processor-intensive compared to training a dense (fully-connected) DNN. In this paper, we utilize Sparse Matrix-Matrix Multiplication (SpMM) to train sparsely-connected DNNs as opposed to dense matrix-matrix multiplication used for training dense DNNs. In our C/C++ implementation, we extensively use in-memory Compressed Sparse Column (CSC) data structures to store and traverse the neural network layers. Also, we train the neural network layer by layer, and within each layer we use 1D-Column partitioning to divide the computation required for training among threads. To speedup the computation, we apply the bias and activation functions while executing SpMM operations. We tested our implementation using benchmarks provided by MIT/IEEE/Amazon HPEC graph challenge [1]. Based on our results, our single thread (1 core) and multithreaded (12 cores) implementations are up to $22\times$, and $150\times$ faster than the serial Matlab results provided by the challenge. We believe this speedup is due to the 1D-Column partitioning that we use to balance the computation of SpMM operations among computing threads, the efficient mechanism that we use for memory (re)allocation of sparse matrices, and the overlapping of the accumulation of SpMM results with the application of the bias and activation functions.**

I. Introduction

Deep Neural networks (DNNs) [2], [3] have influenced different computational fields such as natural language processing and computer vision with their ability to extract useful features within the data across different layers. Traditionally, DNNs' layers are fully (densely) connected where each neuron in one layer is connected to all neurons in the next layer. The full-connectedness characteristic of these networks requires a significant amount of memory and processing power. In addition, utilizing activation functions such as Rectified Linear Unit (ReLU) results in deactivated neurons (having zero value) and sparse outputs for hidden layers which is wasteful if treated densely. Hence, the first question that arises here is "do we really need a dense (fully-connected) DNN?". The answer is **no**, because a sparsely connected DNN has less training complexity and memory requirement while offering comparable accuracy to a dense DNN [4]. Then, the next question is "how to encode the sparsity of hidden layers of a sparse DNN in an efficient way so that we can train the DNN quickly?". The answer to this question is to use compressed sparse formats which offer fast sequential access to sparse matrices [5].

In this work, we introduce a multithreaded implementation that trains sparse DNNs. We utilize C/C++ and OpenMP [6] to parallelize the training and building of DNNs. In addition, we employ Compressed Sparse Column (CSC) [5] to store the layers of the sparse DNNs. Moreover, we use a column by column SpMM algorithm with Sparse Accumulators (SPAs) [5] to train them. In our implementation, the SpMM is done layer by layer, calculating and propagating the weights through the neural network. The SpMM computation done per layer is partitioned using 1D-Column partitioning, with each partition assigned to a unique thread (core). Experiments are done on a machine with a 12-core Intel Xeon CPU (@ 3.40GHz speed). Our multithreaded implementation is up to $150\times$ faster than the provided serial Matlab results from the challenge [1] and up to $140\times$ compared to a serial Matlab running at the same machine where we conducted our experiments.

The rest of this paper is organized as follows. Section II introduces the problem statement. Section III introduces a brief background. Section IV describes the challenges we have faced and overcame in our implementation. In Section V, we discuss the details of our implementation. Section VI describes the experimental setup and reports the performance of our multithreaded implementation. Finally, Section VII concludes the paper.

II. Problem Statement

Neural network layers can be represented using the "triplet format", where a triplet i, j, w implies a connection from neuron i of a layer to neuron j of the following layer with w as the weight of their connection. In its simplest form, training a sparse DNN using forward propagation boils down to an iterative SpMM operation from linear algebra domain as shown in (1):

$$Y_{L+1} = h((Y_L \times W_L) + b_L) \quad (1)$$

where L is the index of hidden layer which is the same as the iteration SpMM is at; Y_0 is the input layer, Y_L is an $n \times m$ input sparse matrix, W_L is the L^{th} $m \times p$ hidden layer matrix, and Y_{L+1} is an $n \times p$ sparse matrix resulting from a previous SpMM. The function h is a nonlinear mapping function such as the ReLU activation function $h(y) = max(y, 0)$. Finally, b_L is a one dimensional column vector of biases for L^{th} layer.

III. Background

A. Deep Learning

Deep learning [2] has delivered promising advancement in many large-scale practical problems such as natural language processing [7], [8], speech recognition [9], [10], and computer vision [11], [12]. Emergence of virtual assistants, self-driving cars, and online item recommendation systems are dramatically accelerated by the research conducted in deep learning. This dramatic change in IT industry is significantly accredited to the research on the scalability of dense neural networks via revamping their architectures. Often, these complex architectures can simply be represented by graphs where the relational representation of graphs makes it possible to exploit graph structures for weight propagation [13].

B. Sparse Matrix-Matrix Multiplication

Distributed training of a sparse deep neural network can essentially be reduced to the problem of parallel execution of matrix - matrix multiplication primitive. The theory of distributed matrix - matrix multiplication spans over decades of research with Cannon's algorithm [14] and Scalable Universal Matrix Multiplication Algorithm (SUMMA) [15] as examples of parallel dense implementations. Gustavson algorithm [16], Sparse Accumulator (SPA) [5], sparse Cannon [17], and Sparse SUMMA [18] are suggested for Sparse Matrix-Matrix Multiplication (SpMM) [19].

SpMM, $C = A \times B$ is a widely used operation, where the results of multiplying two input sparse matrices A and B produces a sparse output matrix C and the dimensions of A, B and C are $n \times m$, $m \times p$ and $n \times p$, respectively [20]. The matrices, A, B, and C are commonly stored using Compressed Sparse Column (CSC), which essentially stores only the nonzero elements [20].

C. Compressed Sparse Column Format

The Compressed Sparse Column (CSC) [5] format stores an $n \times m$ input matrix A using three one dimensional arrays JA, IA, and VA. JA is an array of column pointers, IA is an array of row numbers, and VA is an array that contains the nonzero values (or weights) in A. As such, $|JA| = n+1$, $|IA| = nnz$, and $|VA| = nnz$, where n is the number of rows and nnz is the number of nonzero elements of A. In case of a highly sparse matrix, CSC can significantly save memory, because CSC has space requirement of $n+2nnz+1$, whereas a dense matrix has $n \times m$ space requirement. Also, CSC provides column-major sequential access to A's data which facilitates the SpMM operation as well.

D. 1D-Column Matrix Partitioning

Matrix partitioning divides and distributes a matrix among multiple working threads. Having t threads, 1D-Column partitioning, partitions an $n \times m$ matrix A into t partitions where each partition has n rows and m/t columns. Figure 1a shows how matrix A is tiled into four partitions. During execution, each partition is assigned to a unique thread. Figure 1b shows the assignment of four threads to tiles using 1D-Column.

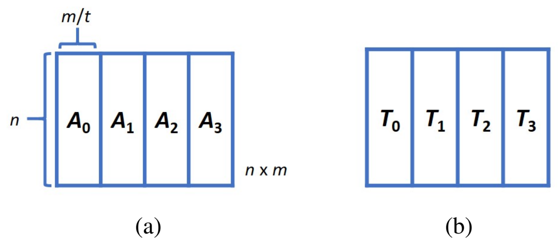

Fig. 1: (1a) 1D-Column partitioning, and (1b) 1D-Column thread assignment for $t = 4$ threads.

IV. SpMM Implementation Challenges

In this section we discuss the challenges we have solved in order to design our multithreaded program for sparse DNN training. These challenges include the SpMM algorithm selection and memory allocation for storing the output matrix.

A. Choice of SpMM Algorithm

Our first challenge to run the SpMM was to find an efficient algorithm to execute $C = A \times B$. Gustavson algorithm [16] is a well-known SpMM algorithm that multiplies rows of A with columns of B and stores the results in rows of C. If we compress A using Compressed Sparse Row (CSR) and B using CSC, we can directly multiply rows of A by columns of B and store the results into a CSR representation of C row by row. The advantage of this algorithm is that C is created row by row at once and we can avoid contention during the accumulation of results. However, this approach is not efficient because for all nonzero entries of i^{th} row of A, we should lookup identical nonzero entries of j^{th} column of B ($\exists\ C_{ij}$ iff $\exists\ A_{i:} \wedge B_{:j}$; : denotes all rows or columns). Thus, the approximated number of lookups is $\sum_{i,j} (nz(A_{i:}) + nz(B_{:j}))$ where $nz()$ returns the number of nonzero elements of i^{th}/j^{th} row/column. This algorithm is clearly not efficient because while multiplying a row of A by a column of B, it needs to exhaustively scan all nonzero entries of those row and column to match the identical entries.

A better way to implement SpMM is to run the multiplication of A and B column by column and store the partial results in a Sparse Accumulator (SPA) [5] which is later used to construct a column of C. In this approach, A and B are stored in CSC formats (to provide column-major access), then columns of B are multiplied by columns of A and the results are accumulated in a SPA vector temporarily. In other words, B_j (j^{th} column of B) is multiplied by all columns of A and the results are gradually accumulated in a SPA as specified by the nonzero rows of B_j. Eventually, after finishing multiplying B_j by A, the SPA will form C_j (j^{th} column of C) [20]. The approximated number of operations for this approach is $\sum_j (nz(B_{:j}) \times (\sum_k nz(A_{:k}) \text{ iff } \exists B_{ij}))$ which is extremely less than the number of lookups of the former described method. Because B_j can precisely index A, avoiding any extra calculation for matching nonzero entries.

B. Choice of SpMM Output Matrix Allocation

The second challenge is to find an upper bound for the size of C. A simple and accurate way to approximate this is to run a symbolic SpMM before running the real SpMM operation [21]. The more complex way is to dynamically reallocate C as SpMM operation executes [22]. The symbolic method requires an extra pass over A and B to calculate the size of C but can allocate the exact required memory, whereas the dynamic method does not need that extra pass, but it imposes memory allocation overhead during SpMM operations which tends to be a performance bottleneck. We choose the former method as we found out this approach is faster than fusing memory allocation with SpMM execution on a multicore machine.

V. Multithreaded Layer-wise Training of Sparse Deep Neural Networks

Basically, training a neural network can be translated into an iterative SpMM, $Y_{L+1} = h((Y_L \times W_L) + b_L)$ where the number of iterations is the number of hidden layers of the neural network, Y_L and W_L are the L^{th} input weights and hidden layer, Y_{L+1} is the results of the SpMM, b_L is the bias vector, and h is the activation function. Note that in our algorithms A, B, and C are aliases for Y_L, W_L, and Y_{L+1} (or Z). Algorithm 1 shows the pseudocode of our multithreaded approach. The two key methods in this pseudocode are SpMM_Sym() that estimates an upper bound for the required memory, and SpMM() that runs the numeric SpMM. In the following, we discuss the details of our implemented approach.

A. Forking and Joining Threads

Before starting the main loop for training the DNN, we use Fork() to launch t threads and finally after the training loop, Join() is used to yield threads. We use OpenMP [6] to support multithreading in our implementation. Algorithm 2 shows how columns of W_L are partitioned and distributed among threads using 1D-Column partitioning. Given, the outer-loop of SpMM() is an independent loop based on the columns of W_L, 1D-Column can break the computation into highly parallelized blocks without need for any concurrency control mechanism while executing the SpMM.

B. Symbolic SpMM

The symbolic SpMM calculates the number of nonzero elements (nnz) results from multiplying Y_L by W_L. Note that each thread has a separate SPA to allow threads accumulate concurrently. As shown in Algorithm 3, in SpMM_Sym() each q^{th} thread iterates over a unique subset of columns of W_L (or B in pseudocode) and intercept the corresponding entries of Y_L (or A in pseudocode). If there are entries that result into nonzero entries, the associated SPA entry will be set to one. Later, each q^{th} thread gathers its SPA while multiplying a column of W_L by the entire Y_L, then updates its local nnz_q, and finally resets the SPA values. After running the symbolic SpMM, the maximum size of SpMM result is calculated. We treat this as maximum number of nonzeros that results of SpMM might have ($nnzmax$) as the activation function might also result into some zero entries. Note that memory can become a huge performance bottleneck, if overallocated. To avoid stressing the memory controller, after estimating the size of Z, the ReInit() method is called to (re)allocate a CSC, e.g., to grow/shrink the size of a currently allocated CSC.

Algorithm 1 Sparse DNN Training $(Y_{L+1} = h((Y_L \times W_L) + b_L))$

```
 1: $Y_0$ is the $n \times m$ input layer CSC
 2: $W$ is $\lambda$, $m \times p$ hidden layer CSCs i.e. $\lambda$ is #layers & $m == p$
 3: $Z$ is an $n \times p$ temporary CSC storing the results of SpMM
 4: $s$ is a $t$, $n \times 1$ 1D row SPA i.e. $t$ is #threads
 5: $b$ is $\lambda$, $1 \times m$ 1D column bias vectors
 6: $h$ is the ReLU activation function
 7: $nnz$ is the number of nonzeros
 8: $nnzm$ is the maximum $nnz$
 9: Fork($t$)
10: for $L = 0 \to \lambda$ do
11:     SpMM_Sym($Y_L$, $W_L$, $Z$, $s_q$, $q$)   ▷ Symbolic SpMM
12:     SpMM($Y_L$, $W_L$, $Z$, $s_q$, $b_L$, $q$)   ▷ Numeric SpMM
13: Join()
```

Algorithm 2 Fork and Join Threads

```
 1: function Fork($t$)
 2:     For-loops convention is For($i = 0;\ i < n;\ i++$)
 3:     for $q = 0$ to $t$ do                ▷ Launch $t$ threads
 4:         fork($q$)                        ▷ $q$ as thread id of $T_q$
 5:         $start\_c_q \leftarrow (p/t) * q$            ▷ Start column
 6:         $end\_c_q \leftarrow start\_c_q + (p/t)$     ▷ End column
 7:         $offset_q \leftarrow 0$                      ▷ zero offset
 8:         $nnz_q \leftarrow 0$
 9:         $nnzm_q \leftarrow 0$                        ▷ Maximum $nnz$
10: function Join( )
11:     join()                                ▷ Join threads
```

Algorithm 3 Symbolic SpMM

```
 1: $A = Y_L, B = W_L, C = Z, s = s_q$
 2: function SpMM_Sym($A, B, C, s, q$)
 3:     $A_{n\times m}$ and $B_{m\times p}$ are CSCs
 4:     $C$ is the temporary empty or already allocated CSC
 5:     $s_{n\times 1}$ is the dense SPA of $T_q$
 6:     for $j = start\_c_q$ to $end\_c_q$ do        ▷ 1D-col part.
 7:         for $k = JA_B[j]$ to $JA_B[j+1]$ do
 8:             $l \leftarrow IA_B[k]$
 9:             for $o = JA_A[l]$ to $JA_A[l+1]$ do
10:                 $s[IA_A[o]] \leftarrow 1$        ▷ Accumulate SPA
11:         for $i = 0$ to $n$ do        ▷ Gather per column of $B$
12:             if $s[i] > 0$ then
13:                 $nnzm_q \xleftarrow{+} 1$
14:                 $s[i] \leftarrow 0$
15:     $nnzm \leftarrow 0$                          ▷ maximum $nnz$
16:     Barrier()
17:     if $q == 0$ then
18:         for $k = 0$ to $t$ do        ▷ Calc. $A \times B$ output size
19:             $nnzm \xleftarrow{+} nnzm_k$
20:         ReInit($Z, m, p, nnzm$)      ▷ (Re)Allocate memory
21:     Barrier()
```

C. Numeric SpMM

The numeric SpMM (SPMM()) executes the real sparse matrix multiplication. The result of this multiplication will be used as an input for the next iteration. In Algorithm 4, each thread iterates over a chunk of W_L columns and column by column multiplies columns of W_L by the entire Y_L, and accumulates the results into its corresponding SPA. After finishing a column of W_L, each thread gathers results from its SPA and stores it into a column of Z (or C in pseudocodes). Later, Z will be copied to Y_{L+1} in order to be used for the next layer. To save future computations, we overlap the SPA gather with adding the bias vector b_L and applying the activation function h. Therefore, we do not need to have another pass over Z to apply these operations. Applying the activation function results into having some zero entries at the end of each partition of Z because each thread pushes nonzero entries from the beginning of its partition. Thus, a refinement step for column pointer of Z is needed so that threads can skip the trailing zero entries of their previous partitions.

D. CSC Refinement

After executing the SpMM, and applying the bias and activation function, Z might have some zero entries because of the threshold applied by the activation function. One way to eliminate these zeros is to recompress and reallocate Z and then copy it to Y_{L+1} (we may skip the copy operation by toggling between Y_{L+1} and Z). The better way which we opt for is to refine Z and then copy the nonzero entries of Z to Y_{L+1}. In Algorithm 5, REFINE() method is called by each thread where it includes updating the column pointer of Z and then calculating an offset to skip the zero entries at the end of each partition of Z. The output of the refinement step is the number of nonzero entries of Z.

E. Copying CSCs

After computing the nnz of Z, thread zero will reallocate Y_{L+1} using REINIT() and then all threads start copying data from their partitions in Z to their partitions in Y_{L+1}. From Algorithm 6, the COPY() method can be run in parallel using the combination of refined column pointers and offsets of zero entries of Z. Therefore, we can construct the compressed Y_{L+1} quickly in parallel.

Finally, the sequence of operations discussed so far from Algorithm 1 are executed by all threads iteratively. In each iteration, new weights (Y_{L+1}s) are computed and propagated through layers of the neural network (W_Ls), and in essence the neural network is trained. Since 1D-Column suits for multithreading of matrix multiplication within each layer, our approach trains the neural network one layer at a time. This offers perfect multithreading per layer where all methods are designed to execute in parallel on their designated partitions.

Algorithm 4 Numeric SpMM

```
1:  $A = Y_L, B = W_L, C = Z$, and $s = s_q, b = b_L$
2:  function SPMM($A, B, C, s, b, q$)
3:      $A_{n\times m}$, $B_{m\times p}$ and $C_{n\times p}$ are CSCs ($JA, IA, VA$)
4:      $s_{n\times 1}$ is the dense SPA of $T_q$
5:      $b_{1\times m}$ is the dense bias vector
6:      for $j = start\_c_k$ to $end\_c_k$ do            ▷ 1D-col part.
7:          for $k = JA_B[j]$ to $JA_B[j+1]$ do
8:              $l \leftarrow IA_B[k]$
9:              for $o = JA_A[l]$ to $JA_A[l+1]$ do
10:                 $s_q[IA_A[o]] \overset{+}{\leftarrow} VA_B[k] \times VA_A[o]$   ▷ Acc.
11:         for $i = 0$ to $n$ do          ▷ Gather per column of $B$
12:             if $s_q[i] > 0$ then
13:                 $s_q[i] \leftarrow h(s_q[i] + b[j])$   ▷ Apply activation
14:                 if $s_q[i] > 0$ then                  ▷ Populate $C$
15:                     $JA_C[j+1] \overset{+}{\leftarrow} 1$
16:                     $IA_C[nnz_q] \leftarrow i$
17:                     $VA_C[nnz_q] \leftarrow s_q[i]$
18:                     $nnz_q \overset{+}{\leftarrow} 1$
19:                     $s_q[i] \leftarrow 0$
20:     Barrier()
21:     $nnz$ = REFINE($Z, q$)          ▷ Refine $Z$'s column pointer
22:     if $q == 0$ then
23:         REINIT($A, n, p, nnz$)
24:     Barrier()
25:     COPY($A, Z, q$)                  ▷ Repopulate $A$ using $Z$
26:     Barrier()
```

Algorithm 5 Refining CSC's Column Pointer

```
1:  $C = Z$
2:  function REFINE($C, q$)
3:      $JA_C[start\_c_q] = 0$
4:      for $k = 0$ to $q$ do
5:          $JA_C[start\_c_q] \overset{+}{\leftarrow} nnz_k$
6:          $offset_q \overset{+}{\leftarrow} (nnzm_k - nnz_k)$
7:      for $j = start\_c_q + 1$ to $end\_c_q$ do
8:          $JA_C[j] \overset{+}{\leftarrow} JA_C[j-1]$
9:      if $q == (t-1)$ then
10:         $JA_C[end\_c_q] \overset{+}{\leftarrow} JA_C[end\_c_q - 1]$
11:     $nnz \leftarrow 0$
12:     if $q == 0$ then
13:         for $k = 0$ to $t$ do
14:             $nnz \overset{+}{\leftarrow} nnz_k$
        return $nnz$
```

Algorithm 6 Copying CSC C to CSC A

```
1:  $A = Y_L, C = Z$
2:  function COPY($A, C, q$)
3:      $JA_A[start\_c_q] \leftarrow JA_C[start\_c_q]$
4:      $nnz_q \leftarrow JA_A[start\_c_q]$
5:      for $j = start\_c_q$ to $end\_c_q$ do
6:          $JA_A[j+1] \leftarrow JA_A[j]$
7:          for $i = JA_C[j]$ to $JA_C[j+1]$ do
8:              $JA_A[j+1] \overset{+}{\leftarrow} 1$
9:              $IA_A[nnz_q] \leftarrow IA_C[i + offset_q]$
10:             $VA_A[nnz_q] \leftarrow VA_C[i + offset_q]$
11:             $nnz_q \overset{+}{\leftarrow} 1$
```

VI. Results

A. Experimental Settings

1) Hardware & Software: Experiments are run on a machine with a 12-core Intel Xeon CPU (@ 3.40GHz speed), and 256 GB memory running Linux OS. Our multithreaded program to train sparse DNNs is open source and freely available to the public[1]. The program is written in C/C++. We extensively use template metrprogramming to support different weight data types. The core data structure we have used is CSC, which is used to compress neural network layers and perform multiplication on them. We have also defined a dense vector data structure to represent bias and SPA vectors. The memory management for all of our key data structures is done by a base allocator class which is backed by `mmap()` with 4KB pages. We also use `mremap()` in order to shrink/grow size of CSCs storing hidden layers after applying the activation function (or when reusing them). We use OpenMP [6] `#pragma omp parallel` (without dynamic task scheduling) to parallelize the symbolic and numeric SpMM execution and CSC recompression precisely based on the 1D-Column partitioning. On the same machine we benchmarked our implementation, we use Matlab R2017a 64-bit [23] to run the serial Matlab code given by the challenge [1] and reported a subset of its results.

2) Datasets: Datasets for the experiments are a set of sparse DNNs provided by the IEEE HPEC challenge [1]. These are synthetic sparse DNNs generated by RadiX-Net synthetic sparse DNN generator [24] with 120, 480, and 1920 layers, and 1024, 4096, 16384, and 65536 neurons per layer, and 32 connections per neuron. The largest dataset has 4.02B connections ($1920 \times 65536 \times 32$). Also, the input to this neural networks is MNIST hadwritten letters [25] which has 60K instances. Furthermore, the correctness is checked using the provided groundtruth data [1] where the nonzero rows of the output layer are tested against the groundtruth data. We have perfect inference score (100% correctness) for all datasets.

B. Runtime Comparison

Table I shows the Matlab results reported in graph challenge draft [1] alongside results collected from running Matlab and our C/C++ implementation on the 12-core Intel Xeon machine. We only ran Matlab for half of the data points, because our Matlab gives comparable results to the results reported in [1]. From Table I, running our developed C/C++ implementation with one thread and 12 threads is up to 22× and 150× faster than the serial Matlab results reported in [1], respectively. Also, with single and 12 threads, we are up to 19× and 140× faster than running Matlab on our 12-core machine.

The right half of Table I shows the strong scaling results, where we increase the number of threads to process the same sparse DNN. From this experiment, running our C/C++ implementation with 12 threads gives up to 7.2× speedup compared to running it with a single thread. This shows our implementation is highly scalable.

[1]The source code is available at https://github.com/hmofrad/SparseDNN

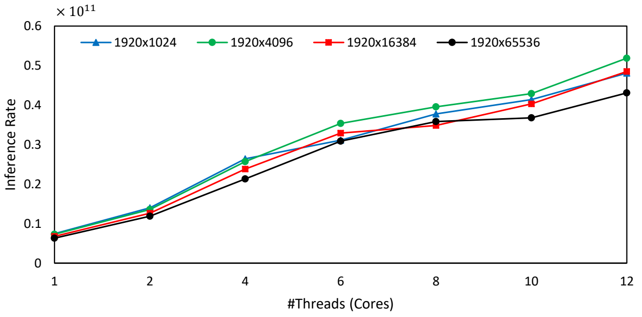

Fig. 2: Inference rate ($\#Inputs \times \#Edges/time$) for different DNN sizes ($\#Layers \times \#Neurons$) and #threads

Here, the speedup against serial Matlab is significant due to the utilized 1D-Column partitioning that breaks the entire computation of each layer between threads while avoiding concurrency control mechanisms. Also, mixing the steps for applying the bias and activation function with SpMM execution helps us skip extra passes over the SpMM output.

C. Inference Rate Comparison

Figure 2 shows the *inference rate* for four DNNs, all with 1920 layers but different number of neurons including 1024, 4096, 16384, and 65536 neurons. The inference rate formula is $\#Inputs \times \#Edges/time$ where $\#Inputs$ is the number of input instances (60,000 for MNIST [25]), and $\#Edges$ and $time$ are the total number of DNN connections, and runtime reported in Table I. From Figure 2, as we increase the number of threads, the inference rate increases as well. From this figure, our implementation has a decent training capacity where by adding more threads, it can effectively train a neural network faster.

VII. Conclusions

Sparse DNNs are reviving the old promise of scalable training of neural networks by providing less memory requirement and computation complexity while offering comparable classification accuracy. In this paper, we leverage SpMM to implement a multithreaded program written in C/C++ that trains sparse DNNs layer by layer. We tested our implementation using MIT/IEEE/Amazon HPEC Sparse DNN challenge datasets [1] and demonstrated that our multithreaded C/C++ implementation outperforms the serial Matlab results provided by the challenge by up to 150×. The 1D-Column partitioning of CSC data structures, efficient memory allocation/reallocation mechanism, and overloaded SpMM execution are among the implemented features that contributed to this performance gain.

The scalability of the current system is limited. It can only scale up within a single machine using multithreading [6]. Moving to a distributed system that uses Message Passing Interface (MPI) [26] for scaling out is among our future directions. Also, the core compression technique which is used in this work is CSC, utilizing better compression techniques such as Doubly Compressed Sparse Column [27] or Triply Compressed Sparse Column [28] is also among our future work.

TABLE I: Runtime of different implementations for Sparse DNN Datasets. First Serial Matlab results are from MIT/IEEE/Amazon HPEC 2019 Challenge [1]. Second Matlab results are from running Matlab on our utilized machine with a 12-core Intel Xeon CPU (@ 3.40GHz). The rest are also from running our multithreaded C/C++ implementation on the same 12-core machine with different number of threads.

Implementation			Matlab [1]	Matlab				C/C++			
#Threads (Cores)			1	1	1	2	4	6	8	10	12
Neurons	Layers	Edges	Time(s)	Time(s)				Time (s)			
1024	120	3932160	626	122.06	37.26	19.37	11.18	9.77	7.37	6.60	5.91
1024	480	15728640	2440	464.50	131.01	69.12	36.86	31.94	23.43	21.74	19.07
1024	1920	62914560	9760	1817.95	506.79	269.27	142.88	121.32	99.93	91.23	77.00
4096	120	15728640	2446	2392.79	145.48	78.63	44.32	30.30	29.24	26.70	23.26
4096	480	62914560	10229	10003.83	529.05	286.47	160.88	111.53	98.82	84.87	75.37
4096	1920	251658240	40245	40464.40	2147.98	1111.61	587.91	427.18	381.85	352.08	291.08
16384	120	62914560	10956		604.03	321.69	183.32	123.63	110.46	102.52	85.18
16384	480	251658240	45268		2067.13	1209.65	638.31	462.00	419.69	380.58	316.55
16384	1920	1006632960	179401		8919.98	4772.50	2536.70	1835.23	1733.85	1498.29	1245.41
65536	120	251658240	45813		2551.63	1422.26	787.85	558.90	472.98	397.48	377.00
65536	480	1006632960	202393		9716.59	5176.92	2910.20	2082.55	1654.41	1516.00	1357.36
65536	1920	4026531840			38260.04	20369.04	11348.16	7823.23	6740.40	6569.96	5341.21

VIII. Acknowledgments

This publication was made possible by NPRP grant #7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). This research was supported in part by the University of Pittsburgh Center for Research Computing through the resources provided. Finally, we thank the IEEE HPEC Graph Challenge 2019 reviewers for their valuable suggestions and comments.

References

[1] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[4] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[5] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[6] OpenMP, "The openmp api specification for parallel programming," https://www.openmp.org/.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[8] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[9] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, 2016, pp. 173–182.

[10] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[11] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[12] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[13] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[14] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University-Bozeman, College of Engineering, 1969.

[15] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[16] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[17] A. Buluc and J. R. Gilbert, *Linear algebraic primitives for parallel computing on large graphs*. University of California, Santa Barbara, 2010.

[18] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[19] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[20] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[21] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 693–702.

[22] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *2012 19th International Conference on High Performance Computing*. IEEE, 2012, pp. 1–10.

[23] Mathworks, "Matlab," https://www.mathworks.com/products/matlab.html.

[24] R. Robinett and J. Kepner, "Radix-net: Structured sparse matrices for deep neural networks," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2019.

[25] Y. LeCun, C. Cortes, and C. J.C. Burges, "The mnist database of handwritten digits," http://yann.lecun.com/exdb/mnist/.

[26] M. Forum, "Mpi forum," https://www.mpi-forum.org/.

[27] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–11.

[28] M. H. Mofrad, R. Melhem, Y. Ahamd, and M. Hammoud, "Efficient distributed graph analytics using triply compressed sparse format," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.