Studying the Effects of Hashing of Sparse Deep Neural Networks on Data and Model Parallelisms

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*—Deep Neural Network (DNN) training and inference are two resource-intensive tasks that are usually scaled out using data or model parallelism where *data parallelism* parallelizes over the *input data* and *model parallelism* parallelizes over the *network*. Also, *dense matrix-matrix multiplication* is the key primitive behind training/inference of *dense DNNs*. On the contrary, *sparse DNNs* are less resource-intensive compared to their dense counterparts while offering comparable accuracy. Similarly, they can be parallelized using data or model parallelism with *Sparse Matrix-Matrix Multiplication (SpMM)* as the key primitive. To scale out, both data and model parallelisms initially use data parallelism to partition the input data among multiple machines. This initial partitioning of the input makes data and model parallelisms performance prone to load imbalance as partitions may be imbalanced. As part of this paper, we take a deeper look into data and model parallelisms and closely study the mechanics of the SpMM used for each. Moreover, to intuitively remedy their load imbalance problem, we incorporate hashing as a simple yet powerful method to address load imabalance. Finally, we use the IEEE HPEC sparse DNN challenge dataset to evaluate the performance of data and model parallelisms at scale. We scaled up to 32 machines (896 cores) and inferred a large sparse DNN with 4B parameters in 51 seconds. Results suggest that with hashing, data and model parallelisms achieve super-linear speedup due to better load balance and cache utilization.**

***Index Terms*—Data parallelism, model parallelism, neural network hashing, sparse matrix matrix multiplication, SpMM**

I. Introduction

Deep Neural Networks (DNNs) are simple yet powerful tools designed to automatically learn features from raw inputs. They are capable of solving complex problems such as computer vision [1], natural language processing [2], and robotics [3] using their simple fully-connected structures. Generally, the learning capacity of a dense DNN is a factor of its number of parameters. However, not all these parameters are contributing to a successful prediction. Therefore, researchers have been studying the effectiveness of sparse DNNs. Sparse DNNs can be generated from either pruning a trained dense DNN [4] or utilizing a sparse architecture from scratch [5]. Although sparse DNNs are sparsely-connected, they have been offering comparable prediction accuracy to their dense ones [5], [6]. The key advantages of sparse DNNs are requiring less memory to store a hypersparse network and executing a smaller number of operations. These make sparse DNNs perfect candidates for training on raw sparse inputs such as video [1], text [2], and sensor readings [7].

As the key algorithm behind the training of DNNs, Back-propagation algorithm [8] comprises of two passes. In the *forward pass*, the algorithm computes the network response for an input and in the *backward pass*, it updates the weights backward using the error calculated for each neuron. Back-propagation algorithm usually uses a variant of gradient decent algorithm [9] to calculate the error for updating the weights backward. The Inference algorithm is identical to the first pass of the DNN training where an input instance is feed to the trained network and the network outputs a prediction. Training/inference shares the same algebraic operations where an input is multiplied by the receptive weights and the summation of their inner products fans out to the connected neurons of the next layer. Therefore, matrix-matrix multiplication turns out be to be the key primitive behind DNN training/inference.

Standard frameworks such as TensorFlow [10], PyTorch [11], and Caffe [12] use dense matrix-matrix multiplication of dense matrices (2D-tensors) [13] for DNN training and inference, while this is not the case for sparse DNNs. Typically, sparse DNNs leverage SpMM where a sparse matrix is compressed using a common sparse matrix compression format [14]. To carry out the training or inference at scale, conventionally data or model parallelism is used where each adopts a variant of the SpMM algorithm. *As part of this paper, we thoroughly investigate the mechanics of SpMM algorithms developed for data and model parallelisms.*

There is abundant of data present today which can be harnessed for machine learning tasks such as machine translation [15], classification [16], and autonomous driving [17]. These tasks are both memory- and compute-intensive that require terabytes of memory and tens of thousands of cores to finish in less than a day. Nowadays, multi-CPU machines are unable to meet the needs of these workloads and even NVIDIA multi-GPU machines [18] can only meet a fraction of such a ginormous configuration. Hence, researchers are apt to utilize data or model parallelism [19], [20] to scale out of a single machine and process the big machine learning workloads at scale. To retool the multi-CPU HPC clusters for massive machine learning tasks, HPC community has been geared toward utilizing MPI [21] or OpenSHMEM [22] open source libraries. In addition, NVIDIA has been developing NVIDIA Collective Communications Library (NCCL) library [23] to meet the same scalability goal for multi-GPU machines.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

In a single machine setting, data and model parallelisms [19], [20] have been widely used to parallelize inference (or training) of DNNs. **Data parallelism** breaks the **input data** into smaller **horizontal partitions** where each partition can be processed separately. As a side effect, data parallelism is more prone to straggler effect when having imbalanced partitions. Examples of data parallelism are TernGrad [24], Mesh-tensorflow [25], and Zero [26]. On the other hand, **model parallelism** breaks the **DNN layers** into **vertical partitions** where these partitions are usually processed following a shared-memory communication Bulk Synchronous Parallel (BSP) scheme [27]. Model parallelism examples are PipeDream [28], Megatron-LM [29], and GPipe [30]. In a distributed setting data and model upscale to **data*data** and **data*model** parallelisms where the input is first partitioned among machines and then data or model is executed on each partition. Therefore, data*data and data*model parallelisms may both suffer from straggler effect due to the input load imabalance among partitions. *As part of this paper, we show how hashing the input or DNN mitigates the effect of stragglers by balancing the computation.*

The rest of this paper is organized as follows. Section II presents a background and surveys the related work. Section III investigates data and model parallelisms. Section IV studies the effect of neural network hashing. Section V reports the results. Finally, Section VI concludes the paper.

II. Background

A. Compressed Sparse Data Structures

Compressed sparse data structures such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) [31], [32] or even optimized variants of them [33], [34] are suitable to represent hypersparse DNNs. Typically, CSR provides sequential row-major access and CSC provides sequential column-major access. These data structures can save on both memory and flops compared to a dense matrix representation. They comprises of three 1-D vectors *IA*, *JA*, and *VA*. In CSR, *IA* is the array of row pointers to where each row begins, *JA* is the array that contains the column indices of nonzero values, and *VA* is the array of the associated nonzero values. In CSC, *JA* is the array of column pointers and *IA* is the array that contains the row indices of nonzero values.

B. Sparse Matrix-Matrix Multiplication

Matrix multiplication $C = A \times B$ [35] is a widely used kernel in many compute-intensive workloads including machine learning, data analytics, and graph computing. Here, $A_{m\times n}$ and $B_{n\times n}$ are first and second input matrices and $C_{m\times n}$ is the output matrix. Also, for an iterative multiplication, C acts as the first input to the next iteration. Cannon's algorithm [13] and Scalable Universal Matrix Multiplication Algorithm (SUMMA) [36] are two well-known dense matrix-matrix multiplication. On the other hand, Gustavson's algorithm [37], sparse Cannon [38], and Sparse SUMMA [39] are examples for SpMM algorithms. Typically, in SpMM, A, B, and, C are stored using a compressed sparse format. Furthermore, as the number of nonzeros of C may change during an iterative SpMM, techniques such as having a *symbolic SpMM step* to estimate the maximum size of C beforehand [40] or *dynamic allocation* of C [41] have been used. In Section III, we will thoroughly investigate two variants of SpMM algorithms designed for data and model parallelisms.

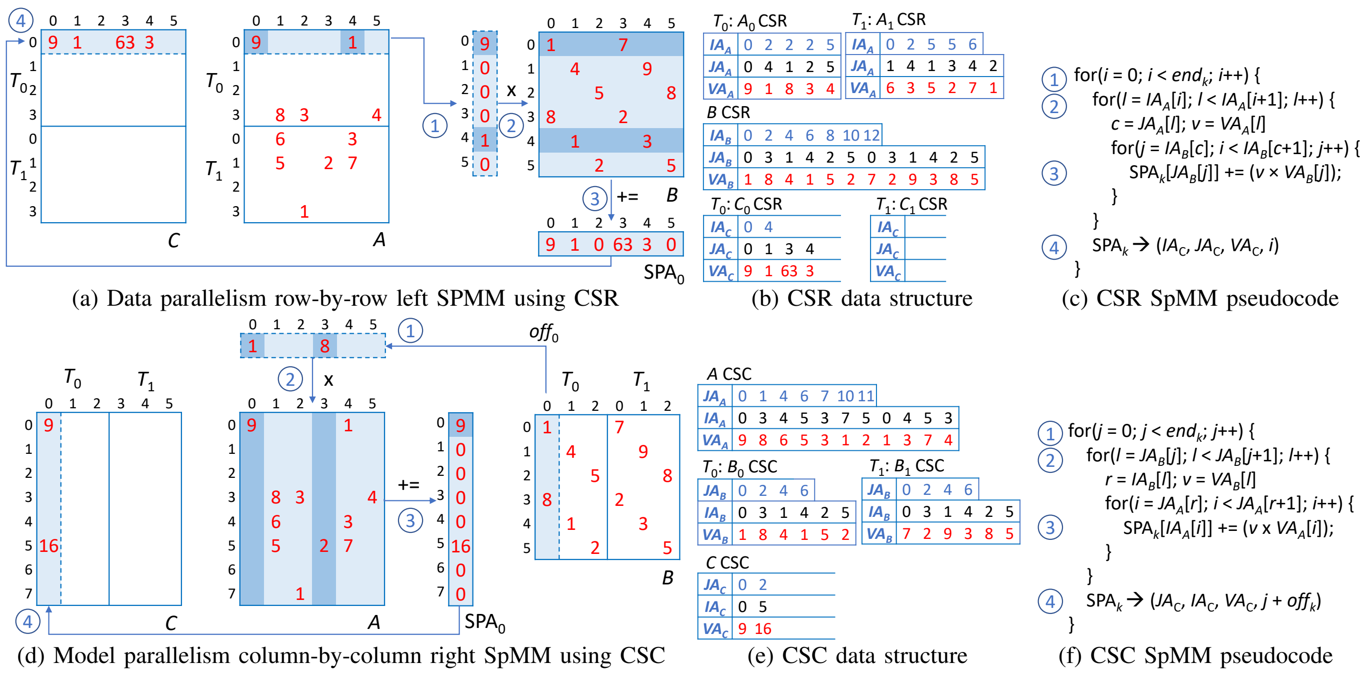
C. DNN Inference in the Language of Linear Algebra

Dense DNNs [42] are composed of fully-connected layers that connect each neuron in one layer to all neurons in the following layer. The core primitive to propagate the weights through dense DNNs is *Dense matrix-matrix multiplication* and their key representation format is *dense matrices*. On the other hand, *sparse DNNs* [43] have sparsely-connected layers that connect each neuron to a subset of neurons in the following layer. Their key primitive is *SpMM* and their key representation format is *compressed sparse matrix format*.

To elaborate, DNN connections can be represented using a *triplet format* from graph theory domain [14], [43], [44], where a triplet (i, j, w) represents a connection from i^{th} neuron of l^{th} layer to j^{th} neuron of $(l+1)^{th}$ layer with w as the weight of this connection. Hence, inference can be represented using the SpMM of $C_{l+1} = h((A_l \times B_l) + b_l)$, where A_l is the l^{th} $m \times n$ sparse input matrix with A_0 being the input layer, B_l is the l^{th} $n \times n$ hidden layer, and C_{l+1} is the $m \times n$ sparse output matrix which is the next input A_{l+1}. The function h is an activation function such as the ReLU $h(y) = max(y, 0)$, and b_l is the bias vector of the l^{th} layer.

D. Data and Model Parallelisms

Data and Model parallelisms [19], [20] are two prominent methods to parallelize DNN training/inference. When having a single process (machine) with t threads, **data parallelism** partitions the $m \times n$ **input matrix** into t **horizontal partitions** of size m/t instances. Also, **model parallelism** partitions the $n \times n$ DNN layers into t **vertical partitions** of size n/t neurons. Data parallelism allows threads to progress independently since each thread processes a separate chunk of the input matrix, whereas, in model parallelism threads are synchronized per layer since partial results produced by each thread should be accumulated before proceedings to the next layer. In a distributed setting, when having p processes each with t threads, **data$*$data** and **data$*$model** parallelisms are used. In these parallelisms, first the input is partitioned into p partitions of size m/p instances alongside the network which is being replicated for each of these partitions. Afterward, in data$*$data each input partition is further broken into t horizontal subpartitions of size $m/(p \cdot t)$, and in data$*$model the network is broken into t vertical partitions of size n/t. Data parallelism inherently suffers from *straggler threads* due to load imbalance among partitions assigned to threads. Similarly, data$*$data and data$*$model parallelisms suffer from *straggler processes* due to load imbalance among partitions assigned to threads. In Section IV, we will show how hashing mitigates the straggler effect for these parallelisms.



(a) Data parallelism row-by-row left SPMM using CSR

T_0: A_0 CSR

IA_A	0	2	2	2	5	
JA_A	0	4	1	2	5	
VA_A	9	1	8	3	4	

T_1: A_1 CSR

IA_A	0	2	5	5	6	
JA_A	1	4	1	3	4	2
VA_A	6	3	5	2	7	1

B CSR

IA_B	0	2	4	6	8	10	12					
JA_B	0	3	1	4	2	5	0	3	1	4	2	5
VA_B	1	8	4	1	5	2	7	2	9	3	8	5

T_0: C_0 CSR

IA_C	0	4		
JA_C	0	1	3	4
VA_C	9	1	63	3

T_1: C_1 CSR

IA_C	
JA_C	
VA_C	

(b) CSR data structure

```
① for(i = 0; i < end_k; i++) {
②     for(l = IA_A[i]; l < IA_A[i+1]; l++) {
          c = JA_A[l]; v = VA_A[l]
          for(j = IA_B[c]; i < IA_B[c+1]; j++) {
③             SPA_k[JA_B[j]] += (v × VA_B[j]);
          }
      }
④     SPA_k → (IA_C, JA_C, VA_C, i)
  }
```

(c) CSR SpMM pseudocode

(d) Model parallelism column-by-column right SpMM using CSC

A CSC

JA_A	0	1	4	6	7	10	11				
IA_A	0	3	4	5	3	7	5	0	4	5	3
VA_A	9	8	6	5	3	1	2	1	3	7	4

T_0: B_0 CSC

JA_B	0	2	4	6		
IA_B	0	3	1	4	2	5
VA_B	1	8	4	1	5	2

T_1: B_1 CSC

JA_B	0	2	4	6		
IA_B	0	3	1	4	2	5
VA_B	7	2	9	3	8	5

C CSC

JA_C	0	2
IA_C	0	5
VA_C	9	16

(e) CSC data structure

```
① for(j = 0; j < end_k; j++) {
②     for(l = JA_B[j]; l < JA_B[j+1]; l++) {
          r = IA_B[l]; v = VA_B[l]
          for(i = JA_A[r]; i < JA_A[r+1]; i++) {
③             SPA_k[IA_A[i]] += (v x VA_A[i]);
          }
      }
④     SPA_k → (JA_C, IA_C, VA_C, j + off_k)
  }
```

(f) CSC SpMM pseudocode

Fig. 1: Parallel SpMM $C = A \times B$ using two threads ($t = 2$, i.e., T_k is the k^{th} thread). (a) - (c) In **data parallelism** matrices are stored in CSR and each thread multiplies a row of A_k by the entire B to produce a row of C_k. (d) - (f) In **model parallelism** matrices are stored in CSC and each thread multiplies a column of B_k by the entire A to produce a column of C.

III. The Duality Between Left and Right SpMM

Gustavson's algorithm [37] is a widely used SpMM algorithm. This algorithm is often combined with other data structures such as Sparse Accumulator (SPA) [32], heap, or hash to produce a row/column of the output matrix C. In the following of this section, we describe Gustavson's left and right SpMM in the context of data and model parallelisms. Note that a symbolic SpMM step to pre-allocate C precedes these SpMM algorithms. Hence, enough memory for C is already allocated.

A. Data Parallelism with Left SpMM

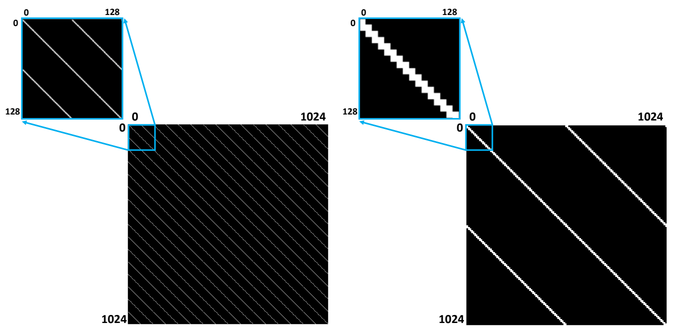
Data parallelism partitions the input A into t partitions where each thread processes a separate partition independently. Since data parallelism horizontally partitions the input instances, a row-major format like CSR perfectly fits this parallelism. Figure 1a depicts the **SPA-based Gustavson's left SpMM algorithm with CSR** for **data parallelism**. In this algorithm, ① each thread T_k extracts a row from A_k (its partition in A), and ② multiplies it by the entire B, ③ while accumulating in SPA$_k$, and ④ finally outputs a row of C_k (its partition in C) by storing the nonzero values of SPA$_k$. Note that C_k acts as the input to the next iteration A_k. Figure 1b shows the CSR representations of A, B, and C where each thread T_k has a separate CSR for its A_k and C_k partitions. Also, B has a single CSR that is shared among threads. Finally, Figure 1c depicts the row-by-row left SpMM algorithm used for data parallelism where end_k is the number of rows in A_k. Note that rows of A_k are re-indexed from 0 to end_k since each partition is allocated separately per thread.

Data parallelism can also be implemented using right SpMM with CSC. However, *at scale* this algorithm is not as efficient as the left SpMM with CSR as it cannot exploit the locality existed in horizontal partitions of data parallelism. Especially, if the partition is balanced and each row is receiving roughly equal number of nonzeros, a row compressed data parallelism like CSR is more efficient. In our experiments, we will compare these two variants of data parallelism.

B. Model Parallelism with Right SpMM

Model parallelism partitions the network B into t partitions where each thread is responsible to execute on a sub-range of columns. The CSC data structure is suitable for model parallelism since this parallelism vertically partitions the network. Figure 1d shows the **SPA-based Gustavson's right SpMM algorithm with CSC** for **model parallelism**. In this algorithm, ① each thread T_k extracts a column of B_k, and ② multiplies it by the entire A_k, ③ while accumulating in SPA$_k$, and ④ finally outputs a column of C_k by storing the nonzeros of the SPA$_k$. Figure 1e shows the CSC format of A, B, and C with B being vertically partitioned among threads. Note that to allow threads randomly access A and C, a single CSC is allocated for each. Figure 1f shows the column-by-column right SpMM algorithm where end_k is the number of columns in B_k and off_k is the offset of B_k from the beginning of B.

Model parallelism can also be implemented using left SpMM with CSR. However, such an implementation requires an extra step to accumulate partial SPAs per row of A which is extremely expensive. So, our discussion on model parallelism is tailored around right SpMM with CSC and we will not explore the left SpMM variant of model parallelism.



(a) An unhashed Radix-Net Layer (b) A hashed Radix-Net Layer

Fig. 2: First layer of A_0 of Table I with white dots as weights. (a) E.g., column ID 1 is only connected to row IDs 1,2, 64, and 65. (b) E.g., column ID 1 is connected to row IDs 1-15.

IV. Neural Network Hashing

A common approach to balance nonzero distribution of a matrix is to hash its rows and columns. Considering an input matrix A and a DNN layer B, hashing can be applied to these matrices in different ways including 1) **Input hashing** which hashes the rows of A. 2) **Layers hashing** which hashes columns of A, and rows and columns of Bs in order to achieve locality in accessing DNN. 3) **Input & layers hashing** which hashes rows and columns of both A and B. Input hashing benefits data*data and data*model parallelisms since it produces balanced input partitions by reordering the input rows. Also, it is a cheap way to mitigate the straggler effect. Furthermore, layer hashing may benefit the SpMM algorithm itself when it yields an optimal access pattern. Hence, a hashing function that provides localized access can effectively benefit the cache hierarchy.

Figure 2a shows the first layer of A_0 DNN of Table I where each column (neuron) has 32 connections. These connections are spread over the entire column where, e.g., first column has connections in row IDs 1, 2, 64, 65, ..., and second column has connections in row IDs 2, 3, 66, 67, ..., etc. Considering model parallelism, this layout leads to an extremely poor access pattern for its right SpMM algorithm because: 1) Those 32 connections are scattered throughout the columns and thus it forces the SpMM algorithm to almost traverse the entire A for each column of B which is expensive. 2) Connections that are placed in each column are different from the ones placed in its next column. Hence, per column the SpMM algorithm should index a completely different set of columns in A. Based on these two characteristics, the original layout of the DNNs generated by Radix-Net [45] is not cache efficient. To address this disadvantage, we use a 2D bucket hashing algorithm [46] to hash rows and columns of the DNN. Figure 2a shows the first layer of A_0 DNN of Table I after its rows and columns are hashed. From this figure, e.g., the 32 connections of column IDs 1-6 are to row IDs 1-15, 512-527, and 1024. So, hashing congregate the connections around the diagonal of the matrix instead of being dispersed within the matrix. This layout is extremely in favor of cache hierarchy because same subsets of contiguous rows of A are recurrently being accessed.

TABLE I: Sparse DNNs dataset [43]. m, n, nnz and L are numbers of instances, features/neurons, nonzeros, and layers, respectively. First column is used as an ID for DNN scale.

	Input		Network			
			Each Layer		All Layers	
ID	Size ($m \times n$)	nnz	Size ($n \times n$)	nnz	L	nnz
A_0	60 K × 1 K	6.3 M	1 K × 1 K	32 K	120	3.9 M
A_1	60 K × 1 K	6.3 M	1 K × 1 K	32 K	480	15.7 M
A_2	60 K × 1 K	6.3 M	1 K × 1 K	32 K	1920	62.9 M
B_0	60 K × 4 K	25 M	4 K × 4 K	131 K	120	15.7 M
B_1	60 K × 4 K	25 M	4 K × 4 K	131 K	480	62.9 M
B_2	60 K × 4 K	25 M	4 K × 4 K	131 K	1920	251 M
C_0	60 K × 16 K	98.8 M	16 K × 16 K	524 K	120	62.9 M
C_1	60 K × 16 K	98.8 M	16 K × 16 K	524 K	480	251 M
C_2	60 K × 16 K	98.8 M	16 K × 16 K	524 K	1920	1 B
D_0	60 K × 65 K	392 M	65 K × 65 K	209 K	120	251 M
D_1	60 K × 65 K	392 M	65 K × 65 K	209 K	480	1 B
D_2	60 K × 65 K	392 M	65 K × 65 K	209 K	1920	4 B

V. Results

A. Experimental Settings

1) Hardware Specifications: A cluster of **32 machines (896 cores)** is used to run our experiments. Each machine has 28-core Intel Xeon CPU @ 2.60GHz and 192 GB memory. Intel MPI [21] is used for building and executing binaries as well as distributing input partitions among machines. Two MPI processes are launched for each machine (one per socket) and Pthread [47] is used to launch threads inside MPI processes.

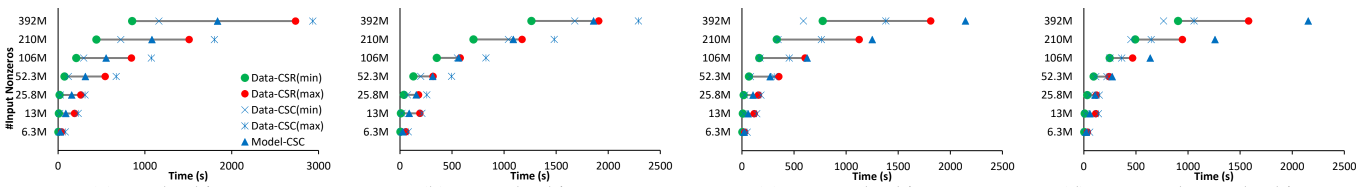
2) Software Specifications: We developed a new DNN inference engine in C++[1] that supports *SPA-based left and right SpMM kernels* which are backed by *CSR, and CSC formats*. These SpMMs consist of two steps including, *symbolic SpMM step* that estimates the size of the output matrix and allocates memory for it, and the *real SpMM step* that runs the SpMM algorithm and generates the output matrix. Leveraging these kernels, we implement *data parallelism* in two flavors of left and right SpMM and *model parallelism* in right SpMM flavor only. At scale, *data*data* and *data*model* parallelisms are used where data parallelism is first used to distribute the input among multiple processes. Last, **2D bucket hashing** [46] with 128 buckets is used to hash the input and/or DNN layers.

3) Datasets: Table I illustrates the IEEE HPEC sparse DNN challenge dataset [43]. This dataset is generated by RadiX-Net sparse DNN generator [45] with 120, 480, and 1,920 layers; 1,024, 4,096, 16,384, and 65,536 neurons per layer, and 32 connections per neuron. The input to these DNNs is MNIST dataset [48] with 60,000 instances and respective number of features (equals to the number of neurons).

B. Single Machine Benchmarking

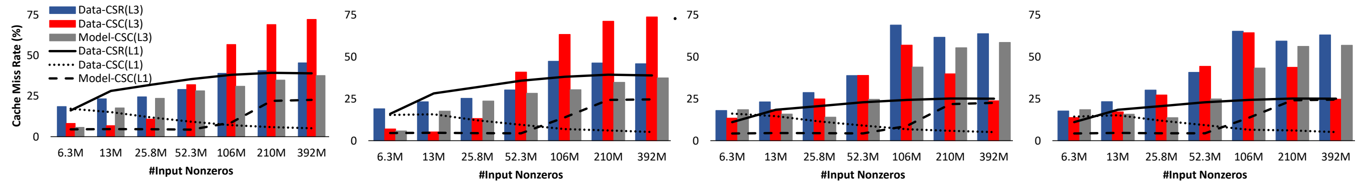
Figure 3 and 4 are the results for left and right SpMM data parallelism with CSR and CSC, and right SpMM model parallelism with CSC on D_2 DNN of Table I using a 28 core machine with $p = 1$ and $t = 28$. The y-axis represents different input sizes from the set of 6.3 M, 13 M, 25.8 M, 53.3 M, 106 M, 210 M, 392 M nonzeros (associated with 1,000, 2,000, 4,000, 8,000, 16,000, 32,000, 60,000 input samples).

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



(a) No hashing (b) Input hashing (c) Layers hashing (d) Input + layers hashing

Fig. 3: Runtime comparison of different parallelisms processing D_2 of Table I on a 28 core machine with $p = 1$ and $t = 28$. (a) - (d) are different hashing types with y-axis as the input size varying from 6.3 M (1,000 sample) to 392 M nonzeros (60,000)



(a) No hashing (b) Input hashing (c) Layers hashing (d) Input + layers hashing

Fig. 4: Cache utilization of different parallelisms processing D_2 of Table I on a 28 core machine with $p = 1$ and $t = 28$. (a) - (c) are different hashing types with x-axis as the input size varying from 6.3 M (1,000 sample) to 392 M nonzeros (60,000).

1) Runtime Variability: Figure 3 reports the effect of different hashing types on runtime of different parallelisms. It presents the runtime variation of data parallelism by showing the min and max runtime associated with the fastest and slowest threads. The variation only exists in data parallelisms as threads can progress independently. According to this figure, the variation escalates when inputs are larger which is due to the load imbalance among threads. Although this property allows some thread to finish early, it creates the undesirable effect of stragglers. On the other hand, the end-to-end runtime does not have any variation in model parallelism since threads should strictly abide synchronization barriers to correctly accumulate the results for each layer.

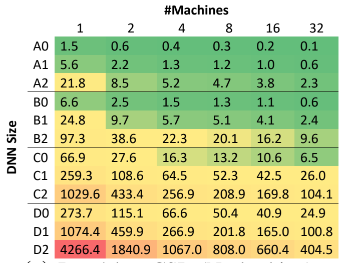
Comparing Figure 3a (no hashing is applied) with Figure 3b (input data is hashed), hashing of the input mitigates the straggler effect by balancing the partitions and hence reducing the variation of runtime in data parallelisms. Input hashing does not affect model parallelism because hashing of the input only reorders the computation of its right SpMM. Moreover, comparing 3a (no hashing) with Figure 3c (layers are hashed), hashing of DNN significantly reduces the end-to-end runtime of CSC-based data parallelism along with its runtime variability. By reordering the rows, layer hashing renders rows together which turns out to be exceptionally suited the right SpMM (see Figure 2b). Last, as shown in Figure 3d, if we apply hashing on both input and layers, the runtime for both CSR and CSC data parallelisms improve.

2) Cache Utilization: Figure 4 shows L1 and L3 cache miss rate of different parallelisms. As a rule of thumb, increasing the number of input instances from left to right should cause cache miss rate to increase due to putting more stress on the cache hierarchy. However, data parallelism with CSC does not conform to this observation when the input data is large enough. We will discuss the reason behind this shortly.

Comparing Figure 4a with Figure 4b, hashing of the input does not affect the cache utilization. To retrace this, we need to have a deeper look into the left and right SpMMs. In left SpMM (data parallelism with CSR), hashing of the input only reordered the input rows and hence it essentially does not alter the nature of the SpMM algorithm. Moreover, in right SpMM (data and model parallelisms with CSC), input hashing does not provide any advantage as it does not change the overall nonzero distribution (count) of the input columns. Finally, a typical use case of input hashing is for data*data and data*model to achieve load balance when scaling out which will be discussed in the next section.

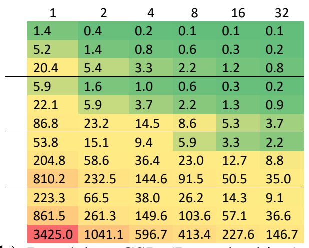
Inherently, w/ or w/o input hashing data parallelism does not have a good L3 performance because each thread can progress independently. Therefore, at any point of time copies of different layers sits in L3 that may be invalidated/evicted shortly by any thread. However, based on Figure 4a and Figure 4b model parallelism has a decent L3 utilization since all threads are accessing a single shared layer matrix. Oddly enough, when layers (Figure 4c) or both input and layers (Figure 4d) are hashed data parallelism with CSC offers superior L3 utilization with a peak utilization at 106 M nonzeros. This phenomenon is highly accredited to its right SpMM that multiplies L1-friendly hashed DNNs by a smaller input partition that fits into L3.

3) Implications of hashing: The left multiplication of data parallelism accesses input rows sequentially and layers rows randomly. This parallelism can benefit from having balanced partitions since balanced partitions (created by hashing) uniformly distribute the input among threads while amortizing the access latency to the DNN rows. Hence, this parallelism has a decent cache utilization. On the other hand, the right multiplication of data and model parallelisms with CSC can highly exploit the underlying structure of DNN (if existed or created by hashing) and boost the cache performance. These parallelisms access the DNN sequentially and the input randomly. Hence, a cache-friendly DNN architecture can perfectly elevate their input's random access pattern to a pseudo-sequential pattern. Last, model parallelism offers better cache utilization for smaller input sizes. This indicates model parallelism would perform better in a distributed setting where many threads process small input partitions. In the next section, we study the scalability of these parallelisms.


#Machines
DNN Size

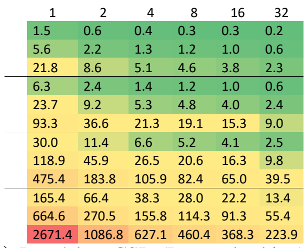
	1	2	4	8	16	32
A0	1.5	0.6	0.4	0.3	0.2	0.1
A1	5.6	2.2	1.3	1.2	1.0	0.6
A2	21.8	8.5	5.2	4.7	3.8	2.3
B0	6.6	2.5	1.5	1.3	1.1	0.6
B1	24.8	9.7	5.7	5.1	4.1	2.4
B2	97.3	38.6	22.3	20.1	16.2	9.6
C0	66.9	27.6	16.3	13.2	10.6	6.5
C1	259.3	108.6	64.5	52.3	42.5	26.0
C2	1029.6	433.4	256.9	208.9	169.8	104.1
D0	273.7	115.1	66.6	50.4	40.9	24.9
D1	1074.4	459.9	266.9	201.8	165.0	100.8
D2	4266.4	1840.9	1067.0	808.0	660.4	404.5

(a) Data*data-CSR (No hashing)



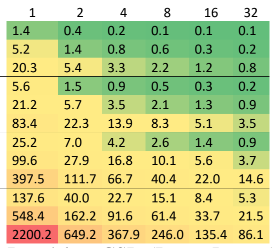
1	2	4	8	16	32
1.4	0.4	0.2	0.1	0.1	0.1
5.2	1.4	0.8	0.6	0.3	0.2
20.4	5.4	3.3	2.2	1.2	0.8
5.9	1.6	1.0	0.6	0.3	0.2
22.1	5.9	3.7	2.2	1.3	0.9
86.8	23.2	14.5	8.6	5.3	3.7
53.8	15.1	9.4	5.9	3.3	2.2
204.8	58.6	36.4	23.0	12.7	8.8
810.2	232.5	144.6	91.5	50.5	35.0
223.3	66.5	38.0	26.2	14.3	9.1
861.5	261.3	149.6	103.6	57.1	36.6
3425.0	1041.1	596.7	413.4	227.6	146.7

(b) Data*data-CSR (Input hashing)



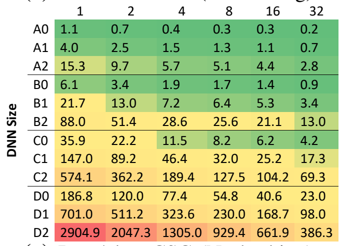
1	2	4	8	16	32
1.5	0.6	0.4	0.3	0.3	0.2
5.6	2.2	1.3	1.2	1.0	0.6
21.8	8.6	5.1	4.6	3.8	2.3
6.3	2.4	1.4	1.2	1.0	0.6
23.7	9.2	5.3	4.8	4.0	2.4
93.3	36.6	21.3	19.1	15.3	9.0
30.0	11.4	6.6	5.2	4.1	2.5
118.9	45.9	26.5	20.6	16.3	9.8
475.4	183.8	105.9	82.4	65.0	39.5
165.4	66.4	38.3	28.0	22.2	13.4
664.6	270.5	155.8	114.3	91.3	55.4
2671.4	1086.8	627.1	460.4	368.3	223.9

(c) Data*data-CSR (Layers hashing)



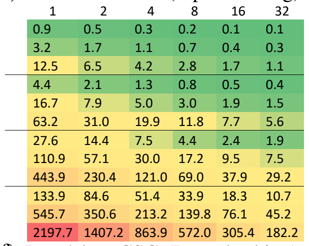
1	2	4	8	16	32
1.4	0.4	0.2	0.1	0.1	0.1
5.2	1.4	0.8	0.6	0.3	0.2
20.3	5.4	3.3	2.2	1.2	0.8
5.6	1.5	0.9	0.5	0.3	0.2
21.2	5.7	3.5	2.1	1.3	0.9
83.4	22.3	13.9	8.3	5.1	3.5
25.2	7.0	4.2	2.6	1.4	0.9
99.6	27.9	16.8	10.1	5.6	3.7
397.5	111.7	66.7	40.4	22.0	14.6
137.6	40.0	22.7	15.1	8.4	5.3
548.4	162.2	91.6	61.4	33.7	21.5
2200.2	649.2	367.9	246.0	135.4	86.1

(d) Data*data-CSR (Input+Layers)



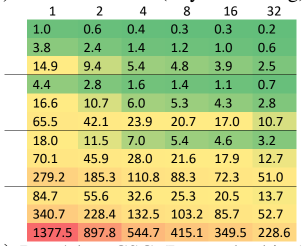
	1	2	4	8	16	32
A0	1.1	0.7	0.4	0.3	0.3	0.2
A1	4.0	2.5	1.5	1.3	1.1	0.7
A2	15.3	9.7	5.7	5.1	4.4	2.8
B0	6.1	3.4	1.9	1.7	1.4	0.9
B1	21.7	13.0	7.2	6.4	5.3	3.4
B2	88.0	51.4	28.6	25.6	21.1	13.0
C0	35.9	22.2	11.5	8.2	6.2	4.2
C1	147.0	89.2	46.4	32.0	25.2	17.3
C2	574.1	362.2	189.4	127.5	104.2	69.3
D0	186.8	120.0	77.4	54.8	40.6	23.0
D1	701.0	511.2	323.6	230.0	168.7	98.0
D2	2904.9	2047.3	1305.0	929.4	661.9	386.3

(e) Data*data-CSC (No hashing)



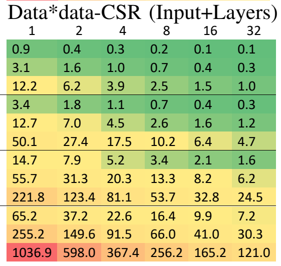
1	2	4	8	16	32
0.9	0.5	0.3	0.2	0.1	0.1
3.2	1.7	1.1	0.7	0.4	0.3
12.5	6.5	4.2	2.8	1.7	1.1
4.4	2.1	1.3	0.8	0.5	0.4
16.7	7.9	5.0	3.0	1.9	1.5
63.2	31.0	19.9	11.8	7.7	5.6
27.6	14.4	7.5	4.4	2.4	1.9
110.9	57.1	30.0	17.2	9.5	7.5
443.9	230.4	121.0	69.0	37.9	29.2
133.9	84.6	51.4	33.9	18.3	10.7
545.7	350.6	213.2	139.8	76.1	45.2
2197.7	1407.2	863.9	572.0	305.4	182.2

(f) Data*data-CSC (Input hashing)



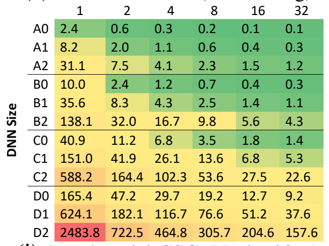
1	2	4	8	16	32
1.0	0.6	0.4	0.3	0.3	0.2
3.8	2.4	1.4	1.2	1.0	0.6
14.9	9.4	5.4	4.8	3.9	2.5
4.4	2.8	1.6	1.4	1.1	0.7
16.6	10.7	6.0	5.3	4.3	2.8
65.5	42.1	23.9	20.7	17.0	10.7
18.0	11.5	7.0	5.4	4.6	3.2
70.1	45.9	28.0	21.6	17.9	12.7
279.2	185.3	110.8	88.3	72.3	51.0
84.7	55.6	32.6	25.3	20.5	13.7
340.7	228.4	132.5	103.2	85.7	52.7
1377.5	897.8	544.7	415.1	349.5	228.6

(g) Data*data-CSC (Layers hashing)



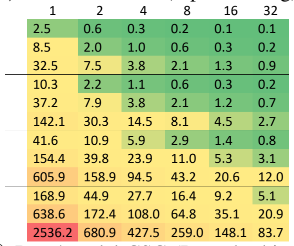
1	2	4	8	16	32
0.9	0.4	0.3	0.2	0.1	0.1
3.1	1.6	1.0	0.7	0.4	0.3
12.2	6.2	3.9	2.5	1.5	1.0
3.4	1.8	1.1	0.7	0.4	0.3
12.7	7.0	4.5	2.6	1.6	1.2
50.1	27.4	17.5	10.2	6.4	4.7
14.7	7.9	5.2	3.4	2.1	1.6
55.7	31.3	20.3	13.3	8.2	6.2
221.8	123.4	81.1	53.7	32.8	24.5
65.2	37.2	22.6	16.4	9.9	7.2
255.2	149.6	91.5	66.0	41.0	30.3
1036.9	598.0	367.4	256.2	165.2	121.0

(h) Data*data-CSC (Input+Layers)



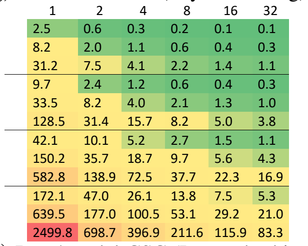
	1	2	4	8	16	32
A0	2.4	0.6	0.3	0.2	0.1	0.1
A1	8.2	2.0	1.1	0.6	0.4	0.3
A2	31.1	7.5	4.1	2.3	1.5	1.2
B0	10.0	2.4	1.2	0.7	0.4	0.3
B1	35.6	8.3	4.3	2.5	1.4	1.1
B2	138.1	32.0	16.7	9.8	5.6	4.3
C0	40.9	11.2	6.8	3.5	1.8	1.4
C1	151.0	41.9	26.1	13.6	6.8	5.3
C2	588.2	164.4	102.3	53.6	27.5	22.6
D0	165.4	47.2	29.7	19.2	12.7	9.2
D1	624.1	182.1	116.7	76.6	51.2	37.6
D2	2483.8	722.5	464.8	305.7	204.6	157.6

(i) Data*model-CSC (No hashing)



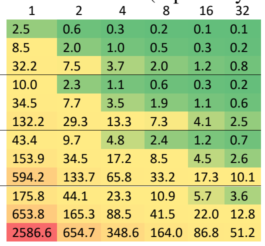
1	2	4	8	16	32
2.5	0.6	0.3	0.2	0.1	0.1
8.5	2.0	1.0	0.6	0.3	0.2
32.5	7.5	3.8	2.1	1.3	0.9
10.3	2.2	1.1	0.6	0.3	0.2
37.2	7.9	3.8	2.1	1.2	0.7
142.1	30.3	14.5	8.1	4.5	2.7
41.6	10.9	5.9	2.9	1.4	0.8
154.4	39.8	23.9	11.0	5.3	3.1
605.9	158.9	94.5	43.2	20.6	12.0
168.9	44.9	27.7	16.4	9.2	5.1
638.6	172.4	108.0	64.8	35.1	20.9
2536.2	680.9	427.5	259.0	148.1	83.7

(j) Data*model-CSC (Input hashing)



1	2	4	8	16	32
2.5	0.6	0.3	0.2	0.1	0.1
8.2	2.0	1.1	0.6	0.4	0.3
31.2	7.5	4.1	2.2	1.4	1.1
9.7	2.4	1.2	0.6	0.4	0.3
33.5	8.2	4.0	2.1	1.3	1.0
128.5	31.4	15.7	8.2	5.0	3.8
42.1	10.1	5.2	2.7	1.5	1.1
150.2	35.7	18.7	9.7	5.6	4.3
582.8	138.9	72.5	37.7	22.3	16.9
172.1	47.0	26.1	13.8	7.5	5.3
639.5	177.0	100.5	53.1	29.2	21.0
2499.8	698.7	396.9	211.6	115.9	83.3

(k) Data*model-CSC (Layers hashing)



1	2	4	8	16	32
2.5	0.6	0.3	0.2	0.1	0.1
8.5	2.0	1.0	0.5	0.3	0.2
32.2	7.5	3.7	2.0	1.2	0.8
10.0	2.3	1.1	0.6	0.3	0.2
34.5	7.7	3.5	1.9	1.1	0.6
132.2	29.3	13.3	7.3	4.1	2.5
43.4	9.7	4.8	2.4	1.2	0.7
153.9	34.5	17.2	8.5	4.5	2.6
594.2	133.7	65.8	33.2	17.3	10.1
175.8	44.1	23.3	10.9	5.7	3.6
653.8	165.3	88.5	41.5	22.0	12.8
2586.6	654.7	348.6	164.0	86.8	51.2

(l) Data*model-CSC (Input+Layers)

Fig. 5: Runtime (in seconds) of different parallelisms on DNNs of Table I using 1 to 32 machines.

C. *Wide-scale Benchmarking*

Figure 5 shows the results of data*data parallelism (CSR & CSC) and data*model parallelism (CSC) on DNNs reported in Table I. X-axis represents the number of machines (cluster scalability) and y-axis represents the DNN size (data scalability). Results are shown using heatmaps to improve data visualization. From this figure, data*data with CSR performs best for smaller DNNs (A_0 to B_2), whereas, data*model with CSC produces the best results for larger DNNs (C_0 to D_2).

Figure 5a - 5d shows the result for data*data parallelism with CSR. For D_2 with 32 machines (right bottom corner), input, layers, and input & layers hashing offer 2.6×, 1.7×, and 4.7× speedups over the unhashed results, respectively. This suggests input hashing improves the runtime significantly and its improvement is even reinforced if combined with layers hashing. Moreover, Figure 5e - 5h are the results for data*data parallelism with CSC. For D_2 with 32 machines, input, layers, and input & layers hashing offer 2.1×, 1.7×, and 3.2× speedups over the unhashed results, respectively. This parallelism is not as scalable as the CSR variant due to its poor cache efficiency when input partitions are small. Figure 5i - 5l shows the results obtained from data*model parallelism. For D_2 with 32 machines, input, layers, and input & layers hashing offer 1.9×, 1.9×, and 3× speedups over the unhashed results, respectively. Both input and DNN hashing can improve the runtime of this parallelism, however, if combined they can offer a significant runtime improvement.

Different speedup trends can be observed if input and/or DNN are hashed. These effects can be explained in terms of cache performance and load imabalance. A **super-linear speedup** occurs when the number of machines is small and hence cache subsystem is under severe pressure. In this case doubling the number of machines results in more than doubling of the speedup since more cache is available. Conversely, when the number of machines is large, the cache conflict is less hence doubling the number of machines does not have a significant effect on the cache conflict and speedup. Therefore, a **sub-linear speedup** happens when the number of machines is large and the effect of load imbalance kicks in and become dominant (whilst cache conflict is no longer dominant).

VI. Conclusion

DNN inference is an embarrassingly parallel compute- and memory-intensive task. Data and model parallelisms can be leveraged to run the inference at scale. In this paper, we thoroughly investigate the internals of data and model parallelisms by focusing on their core SpMM kernels. In addition, we study the effects of hashing on the performance of these parallelisms. We use IEEE HPEC sparse DNN challenge dataset to test these parallelisms on a cluster of 32 machines (896 cores). Our results suggest data parallelism is suitable for smaller DNNs and model parallelism for larger ones. We find out input and layers hashing improve load balance and cache utilization, respectively. Lastly, we observe that these parallelisms can achieve super-linear speedup by hashing the DNN layers.

VII. Acknowledgments

This work is supported by NSF grant CCF-1725657, NPRP grant #7-1330-2-483 and by the University of Pittsburgh Center for Research Computing through the resources provided.

References

[1] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[3] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[4] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[5] 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.

[6] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119*, 2017.

[7] A. Murad and J.-Y. Pyun, "Deep recurrent neural networks for human activity recognition," *Sensors*, vol. 17, no. 11, p. 2556, 2017.

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

[9] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[10] Google, "Tensorflow machine learning platform," https://www.tensorflow.org/.

[11] PyTorch, "Pytorch machine learning framework," https://pytorch.org/.

[12] BerkeleyVision, "Caffe deep learning framework," https://caffe.berkeleyvision.org/.

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

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

[15] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[16] T. K. R. Medini, Q. Huang, Y. Wang, V. Mohan, and A. Shrivastava, "Extreme classification in log memory using count-min sketch: A case study of amazon search with 50m products," in *Advances in Neural Information Processing Systems*, 2019, pp. 13 265–13 275.

[17] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 621–11 631.

[18] NVIDIA, "Nvidia dgx-2," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf.

[19] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[20] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc, "Integrated model, batch, and domain parallelism in training neural networks," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 77–86.

[21] Intel, "Intel mpi library," https://software.intel.com/en-us/mpi-library.

[22] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[23] NVIDIA, "Nvidia nccl," https://developer.nvidia.com/nccl.

[24] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in neural information processing systems*, 2017, pp. 1509–1519.

[25] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 414–10 423.

[26] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimization towards training a trillion parameter models," *arXiv preprint arXiv:1910.02054*, 2019.

[27] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[28] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[29] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv*, pp. arXiv–1909, 2019.

[30] 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.

[31] N. E. Gibbs, W. G. Poole Jr, and P. K. Stockmeyer, "A comparison of several bandwidth and profile reduction algorithms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 2, no. 4, pp. 322–330, 1976.

[32] 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.

[33] 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.

[34] 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.

[35] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on intel knl and multicore architectures," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, 2018, pp. 1–10.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[43] 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.

[44] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[45] 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.

[46] Y. Ahmad, O. Khattab, A. Malik, A. Musleh, M. Hammoud, M. Kutlu, M. Shehata, and T. Elsayed, "La3: A scalable link-and locality-aware linear algebra-based graph analytics system," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, pp. 920–933, 2018.

[47] Linux, "Posix thread (pthread) library," http://man7.org/linux/man-pages/man7/pthreads.7.html.

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