DenseStream: A Novel Data Representation for Gradient Sparsification in Distributed Synchronous SGD Algorithms

Guangyao Li
Institute of Automation, Chinese Academy of Sciences
Beijing, China
liguangyao2020@ia.ac.cn

Mingxue Liao*
Institute of Automation, Chinese Academy of Sciences
Beijing, China
mingxue.liao@ia.ac.cn

Yongyue Chao
Institute of Automation, Chinese Academy of Sciences
Beijing, China
chaoyongyue2020@ia.ac.cn

Pin Lv
Institute of Automation, Chinese Academy of Sciences
Beijing, China
pin.lv@ia.ac.cn

***Abstract*—Distributed training is widely used in training large-scale deep learning model, and data parallelism is one of the dominant algorithms. Data-parallel training has additional communication overhead, which greatly affects the training at low bandwidth. Gradient sparsification is a promising technique to reduce the communication volume, which keeps a small number of important gradient values and sets the rest to zero. However, the communication of sparsified gradients suffer from scalability issues for (1) the communication volume of the AllGather algorithm, which is commonly used to accumulate sparse gradients, increases linearly with the number of nodes, and (2) sparse local gradients may return dense due to gradient accumulation. These issues hinder the application of gradient sparsification. We observe that sparse gradient value distribution has great locality, and therefore we propose DenseStream, a novel data representation for sparse gradients in data-parallel training to alleviate the issues. DenseStream integrates an efficient sparse AllReduce algorithm with the synchronous SGD (S-SGD). Evaluations are conducted by real-world applications. Experimental results show that DenseStream achieves better compression ratio at higher densities and can represent sparse vectors with a wider range of densities. Compared with dense AllReduce, our method is more scalable and achieves 3.1-12.1x improvement.**

***Keywords*—deep learning, AllReduce, gradient sparsification, data representation**

I. INTRODUCTION

Training deep learning models is quickly becoming a major workload on large-scale computing systems. The size of the models and the computation required for training increases significantly. While AlexNet [15] with millions of parameters requires days on a GPU for training, newer models such as GPT-3 [3] which parameters up to 100 billion, take more than hundreds of years to train on a single GPU. Utilizing a cluster of GPU resources for training large-size models is a common practice. There are many parallelization approaches for accelerating training on multiple GPUs [24], and data parallelism is one of the dominant techniques. In data-parallel training, computing nodes maintains consistent model weights. After computing local gradients on each node with different mini-batch samples, the global gradients are accumulated across all nodes through network communication. Model copies are kept in sync by applying global gradients to update the model on all nodes.

Compared with the training process on a single node, data-parallel training is broken down into two phases, calculating the local gradients on each node in parallel and exchanging the parameters or gradients of the model. Data-parallel training brings additional network communication overhead. The training speedup ratio depends on communication-to-computation ratio. If the time of communication is much less than computation, the nodes execute computing tasks without break, computing resources are fully utilized, and the scaling efficiency is close to 1, which is our expectation. On the contrary, when communication takes much longer than computation time, nodes have to wait for communication to complete before starting the next iteration. Computing resources are idle for a long time, and the scaling efficiency is extremely low. Multi-node training may even be slower than single-node. Therefore, to improve the performance of data parallel training, we can either increase the workload of nodes to increase computation time or reduce the communication time in each iteration.

It is difficult to change the neural network model for training, so the way to increase the computation time is to increase the mini-batch size [7][18][25]. However, the GPU memory is limited, and the mini-batch size has an upper bound, so the computation time also has an upper bound. Reducing communication time is more effective. The communication overhead depends on the communication volume and network bandwidth. In classical data parallelism, nodes exchange their gradients or model parameters by AllReduce [8] operation. In model training of deep neural networks, the communication volume is usually very large, and the network bandwidth is limited for some institutions or individuals. Meanwhile, network bandwidth is growing much slower than the model size. Network communication has become the main bottleneck of training.

Therefore, reducing communication volume is the key way to conquer the communication challenge. Very recently, many techniques like gradient sparsification, quantificaiton and compression methods [4][6][11][12][13][26] have been proposed to reduce the volume of network communication. Among these, gradient sparsification [1][19][26] is the promising technique. The main idea is that not all gradient values are equally important, and usually up to 99% values contribute few in each step for model convergence. In other word, there is no significant loss of accuracy with 1% important gradient values preserved, while others set to zero.

* Corresponding author

The local gradients can be sparsified significantly to just about 1% density (99% gradient values are zeros) with gradient sparsification techniques. This benefit the process of gradient accumulation in distributed training since sparse vectors can be highly compressed which reduce the communication volume to break the communication bottleneck.

However, sparse communication still suffers from scalability issues. Specifically, while local gradients are constructed into a very sparse vector with gradient sparsification at each node, the sparsity of global gradients is not guaranteed. If the indexes of non-zero gradient values between nodes do not overlap, as the number of nodes in the cluster increases, the global gradients will quickly become dense, which is called the fill-in problem, and the AllGather-base sparse reduction [5][17][22] makes the issue worse, the communication bottleneck will reappear. And the commonly used key-value data representation requires additional storage of index, when the density exceeds 50%, the additional index storage overhead will cause the overall overhead to exceed the original dense representation (assuming the index and value storage overhead is the same). The data representation might be changed in training. These issues should be considered in application of sparse communication.

In this paper, we focus on the data representation and explore how to represent sparse gradient data more efficiently in deep learning. We propose a sparse data representation algorithm called DenseStream to alleviate the fill-in problem and provide consistent format at arbitrary densities. We implement the DenseStreamAllreduce (DSA) which provides more efficient sparse gradients aggregation from distributed nodes. Then we integrate our proposed DSA to S-SGD under PyTorch and MPI. The DSA S-SGD achieves 3.1-12.1x speedup than dense S-SGD. Compared to high-performance sparse communication library SparCML [5], DSA S-SGD is generally around 1.1 times faster on evaluated experiments. Our contributions are as follows:

- We observed that the *top-k* gradient values clustered in some regions during neural network training.
- We propose an efficient data representation of sparse gradients for neural network training, called DenseStream, and a corresponding sparse AllReduce algorithm, to reduce the communication volume and alleviate the fill-in problem in data-parallel training.
- DenseStream and corresponding AllReduce algorithm achieves improved compression and communication efficiency on the real-world application.

We compared DenseStream with other algorithms on VGG16 [14] and ResNet50 [9]. DenseStream showed good performance, it has a good effect on improving the training speed of the model. We hope that the performance of the algorithm can be tested on a large-scale cluster in the future.

II. Background and Motivation

A. Mini-batch SGD

Stochastic gradient descent is one of the most used algorithms for training deep learning models. Let m be the size of the mini-batch, w_t the parameters of the neural network model at the t-th iteration, (x_i, y_i) a sample and its label in the batch, and L the loss function. Each round of training has two stages. In the forward propagation stage, the current model parameters and samples in the mini-batch are used to calculate the loss through the loss function L. Then in the back propagation stage, the gradient of each trainable parameter is calculated by:

$$G_t(w_t) = \frac{1}{m}\sum_{i=0}^{m} \nabla L(w_t, x_i, y_i) \quad (1)$$

The model parameters update as $w_{t+1} = w_t - \alpha G_t$, where α is the learning rate.

B. Synchronous SGD

Synchronous SGD is widely used in data parallel strategy. It is mainly to expand the SGD to adapt to the distributed environment. In a distributed environment, each node calculates its gradients locally, then the global gradients accumulated by the local gradients are used to update the model. The update formula of parameters is:

$$w_{t+1} = w_t - \alpha \sum_{i=1}^{P} G_i(w_t) \quad (2)$$

where P is the number of computing system nodes. The local gradients are located in different nodes, so that the accumulating operation involves communication costs.

In model training, the size of local gradients and global gradients are the same as the size of the trainable parameters because each trainable parameter needs to be updated in each round of training. If the cluster has a total of P nodes, the global gradients is accumulated through Ring-AllReduce, leading to about $2(P-1)N/P$ communication volume, where N is the amount of trainable parameters of the model. In large-model training, gradients accumulation becomes the bottleneck due to bandwidth constraints.

C. Top-k Sparsification

Algorithm 1 S-SGD with *top-k* sparsification

Inputs: dataset D, initialized weights w, mini-batch size m, iterations T, learning rate α, the number of worker P, the number gradients to select k.

1: Initialize $G_0^i = 0$
2: **for** $t = 1 \rightarrow T$ **do**
3: Sampling a mini-batch of data D_t^i from D;
4: $G_t^i = G_{t-1}^i + \nabla L(w_t, D_t^i)$
5: Select threshold thr = the k^{th} largest value of $|G_t^i|$;
6: $Mask = |G_t^i| > thr$;
7: $G_{local}^i = G_t^i \odot Mask$;
8: $G_t^i = G_t^i \odot \neg Mask$; // The residuals of gradients
9: G_{global}^i =TopKAllReduce(G_{local}^i)
10: $w_{t+1} = w_t - \alpha G_{global}^i$
11: **end for**

Gradient sparsification is a key approach to reduce the communication volume. A commonly used sparsification algorithm is *top-k* sparsification [26]: each node transfers only the k largest (in terms of the absolute value) of its gradient values. The remaining gradients in each node that do not participate in current accumulation will be added to the local

gradients in the next step of training. The global gradients obtained by accumulating sparse local gradients is used for the model update of this iteration. The convergence of *top-k* S-SGD algorithm has been proved [1][26]. However, reaching low density levels (less than 1%) requires extremely careful tuning of hyperparameters, which introducing instability to the training [5]. Employing higher density levels, 5-10% per node, which tends to be more robust, is a better option. The pseudo-code of *top-k* sparsification S-SGD is shown in Algorithm 1.

The fill-in is one of the major obstacles to apply the gradient sparsification on large-scale clusters. Supposing that the model has 10 million trainable parameters, corresponding to 10 million gradient values, and the density is 1%, the number of non-zero local gradient values for each node is 100,000. If there is no overlap index between the non-zero values of local sparse gradients across nodes, the training system with 100 computing nodes will make the global gradient completely dense, and the communication bottleneck appears again. Although there are overlap indexes between local sparse gradients, as the number of nodes in the system grows, the fill-in quickly diminish the benefits of gradient sparsity [5].

III. METHOD

In this section, we first demonstrate our proposed data representation for sparse gradients based on the characteristics of the deep learning model training, and then present the corresponding communication algorithm.

A. Data Representation

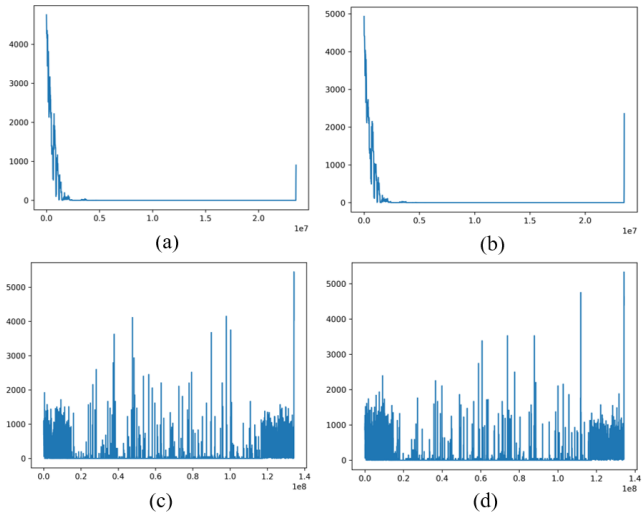

Fig. 1. The histogram of *top-k* gradients with 1% density.The bin width is 10000 indexes. We count the frequency of *top-k* gradients in it at iteration 1 and 5000. (a) and (b) are the results of ResNet50, (c) and (d) are the results of VGG16

Our data representation method is based on our observations that during the training process of the deep learning model, the position distribution of the *top-k* gradient values is not uniform, and peaks appear in some regions. The *top-k* values are more concentrated in some local areas. For example, in the early step of CNNs training, the *top-k* values are concentrated in the shallow convolution kernel, while in the later step they are concentrated in the deep convolution kernel and classification layer. As is shown in Fig. 1, both ResNet50 and VGG16 show the locality. In some regions, nearly half of the values are the *top-k* value. Therefore, during training, some dense regions appear in the sparse gradient vectors. In these dense regions, the indexes of *top-k* values are very close, or even adjacent. It is more efficient to use dense representation instead of key-value representation in the dense regions. However, the position and size of dense region cannot be predicted in advance, so an additional index and length value is required for each dense region to store the information. Therefore, we design the data representation method which we called DenseStream. The basic unit of the DenseStream is a dense block, which consists of three parts: the index for starting position of dense region, the length for the number of elements in dense region, and the gradient values of entire dense region. Let N be the size of the gradient vector, and the storage cost of each gradient value is K bytes, then the dense region is defined as:

1) There allows zero elements, but both the first and last elements of the dense region are non-zero.

2) The number of consecutive zero elements in the dense region does not exceed $\lceil 2\lceil \log_2 N/8 \rceil / K \rceil$.

The reason for *2)* is that, if the number of consecutive zero elements exceeds $\lceil 2\lceil \log_2 N/8 \rceil / K \rceil$, the dense block will be divided into two dense blocks here, which overhead of added index and length is less than that of consecutive zero elements. Therefore, the number of consecutive zero elements is not exceed $\lceil 2\lceil \log_2 N/8 \rceil / K \rceil$.

Assuming that the number of non-zero elements in the vector is *nnz*, the storage overhead of DenseStream is:

$$\left(2\left\lceil\frac{log_2 N}{8}\right\rceil + K\right)\cdot nnz \leq cost_{total} \leq 2\left\lceil\frac{log_2 N}{8}\right\rceil + K\cdot nnz \quad (3)$$

When the vector is very sparse, the overhead is close to the lower bound, which is $\lceil (\log_2 N)/8 \rceil \cdot nnz$ higher than the key-value representation. As $nnz << N$, there will be no bottleneck, and it is acceptable. When the vector is relatively dense, the performance is better than key-value representation. And when the vector is completely dense, there is only one index and length additional overhead than the dense representation, which can be ignored. Therefore, DenseStream is a consistent data representation algorithm with a wider range of density representation.

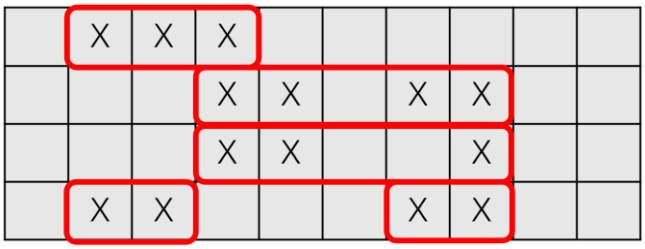

Fig. 2. Examples of dense block. Assuming that the storage cost of index, length, and each element are 4 Bytes. *X* represents the top-k gradient value. The red box is a dense block. The block 1 has no zero element, block 2 has a zero element, block 3 has two consecutive zero elements. These are three consecutive zero element between block 4 and block 5, so these are two blocks.

B. Communication Algorithm

AllReduce is the most important algorithm in data-parallel training, because global gradients are accumulated through AllReduce. The AllReduce algorithm of sparse vector is more complicated than that of dense vector [5]. One reason is that the dense AllReduce algorithm does not perform well on

sparse vectors. Due to the special representation, sparse vectors cannot directly participate in calculations, which increases the overhead of most dense AllReduce algorithms.

Another reason is that the *nnz* of the result vector in sparse AllReduce cannot be predetermined. If there is no overlap between the sparse vectors across different nodes, the *nnz* of result vector will be $P \cdot k$, where k is the *nnz* of sparse vector in a node. But there is a large degree of overlap between sparse vectors across nodes. The *nnz* of result vector is unpredictable. A solution is to use the AllGather instead of AllReduce. However, the communication volume of the AllGather increases linearly with the number of nodes, which is unacceptable in large-scale clusters. For this, we design and implement a communication algorithm suitable for DenseStream.

Our communication algorithm mainly includes three phases: (1) *balanced split*, (2) *scatter-reduce*, (3) *allgatherv*. In the *balanced split* phase, we balanced split the vector dimension N into P partitions and each node is assigned a partition to accumulate the corresponding global gradients. In the *scatter-reduce* phase, each node accumulates the partial global gradients through reduce, and then each node obtains the global gradients of the corresponding partition. In the *allgatherv* phase, each node receives the complete global gradients through AllGatherv.

We assume bidirectional, direct point-to-point communication between the nodes, and consider the classic Latency-Bandwidth cost model. The cost of sending a message of size L is $\alpha+\beta L$, where α is the latency of a message transmission, β is the transfer time per word, and L is the message size in words.

1) balanced split

Balanced split should try to make the partial global gradients accumulated by each node after *scatter-reduce* phase similar in size, which is beneficial to *scatter-reduce* and *allgatherv* phase. But we cannot predict the result of *scatter-reduce* phase, our *balanced split* algorithm is based on current information.

Since the non-zero values of local gradients is not evenly distributed on the gradient vector, uniformly splitting the space dimension into P partitions cannot make the workload balance of each node. In an extreme case, all the non-zero values are in region i of each node, then node i should receive $(P-1)\cdot k$ elements in *scatter-reduce* phase and broadcast it to other nodes in the *allgatherv* phase.

The ideal *balanced split* allows each node to get k/P global gradients after *scatter-reduce*, and each node only receives $(P-1)k/P$ elements. We adopt the *balanced split* algorithm from [16]. Each node selects the *top-k* local gradient values, then sorts them by index, splits them evenly into P partitions, obtains the boundaries. Use AllReduce to average the boundaries of all nodes to get the global boundaries as the result of *balanced split*.

2) scatter-reduce and allgatherv

In the *scatter-reduce* phase, we adopt the recursive halving technique [20]. The behavior is illustrated in Fig. 3. In the first round, nodes with distance $P/2$ apart exchange their $P/2$ partitions and perform a local reduction. In the second round, nodes with distance $P/4$ apart exchange their $P/4$ reduced partitions. Following the pattern, in the $\log_2(P)$ round, nodes with 1 distance apart exchange 1 reduced partitions, and each node gets the partial global gradients.

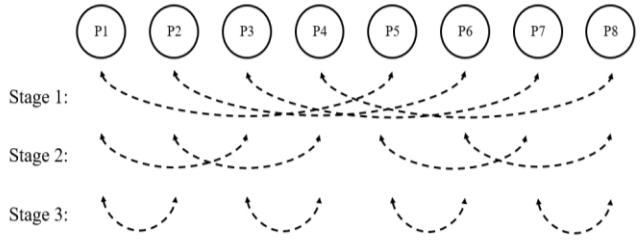

Fig. 3. Recursive halving algorithm with 8 nodes.

Algorithm 2 S-SGD with DenseStream sparsification

Inputs: dataset D, initialized weights w, mini-batch size m, iterations T, learning rate α, the number of worker P, the number gradients to select k.

Initialize $G_0^i = 0$
for $t = 1 \rightarrow T$ **do**
  Sampling a mini-batch of data D_t^i from D;
  $G_t^i = G_{t-1}^i + \nabla L(w_t, D_t^i)$;
  Select threshold thr = the k^{th} largest value of $|G_t^i|$;
  $Mask = |G_t^i| > thr$;
  $G_{local}^i = G_t^i \odot Mask$;
  $G_t^i = G_t^i \odot \neg Mask$; // The residuals of gradients
  G_{local}^i = DenseStream(G_{local}^i); // Compress the local gradients
  G_{global}^i = DenseStreamAllReduce(G_{local}^i);
  Decompress G_{global}^i;
  $w_{t+1} = w_t - \alpha G_{global}^i$
end for
procedure DenseStreamAllReduce(G_{local}^i)
  local_boundaries = balanced_split(G_{local}^i)
  global_boundaries = AllReduce(local_boundaries)
  G_{region}^i = ScatterReduce(G_{local}^i)
  G_{global}^i = AllGatherv(G_{region}^i)
  return G_{global}^i
end procedure

Due to the *balanced split*, each partition contains approximately k/P elements. The *scatter-reduce* latency is:

$$\log_2(P)\alpha + \frac{P-1}{P}k\beta \le latency_{scatter-reduce} \le \log_2(P)\,\alpha + \frac{\log_2(P)}{2}Pk\beta \quad (4)$$

There are $\log_2(P)$ stages, and the latency of a message transmission is $\log_2(P)\alpha$. When the k indexes fully overlap, the latency reaches lower bound, since the non-zero elements of each partition maintain constant size k/P. The upper bound is reached when the indexes do not overlap at all. Therefore, the number of partitions transmitted per round is halved, but the number of elements in a partition is doubled, and communication volume maintains constant size $P/2$.

TABLE I. COMPRESSION RATIO OF DENSESTREAM TO KEY-VALUE

Model	density 1%			density 5 %			density 10%		
	epoch 1	*epoch 5*	*epoch 50*	*epoch 1*	*epoch 5*	*epoch 50*	*epoch 1*	*epoch 5*	*epoch 50*
VGG16	1.11	0.99	1.03	0.86	0.89	0.85	0.75	0.77	0.75
ResNet50	0.98	1.09	1.11	0.86	0.89	0.94	0.82	0.85	0.89

In *allgatherv* phase, we adopt the recursive doubling technique. The behavior is like recursive halving technique: in t round, nodes with distance $P/2^t$ apart exchange their 2^t partitions which contain partial global gradients.

The *allgatherv* latency is:

$$\log_2(P)\alpha + \frac{P-1}{P}k\beta \le latency_{allgatherv} \le \log_2(P)\,\alpha + (P-1)k\beta \quad (5)$$

The latency of a message transmission is $\log_2(P)\alpha$. The lower bound is reached when the indexes full overlap, each node gets k/P elements of global gradients in corresponding partition. And when the indexes do not overlap, each node gets k elements in their partition. Then the latency reaches upper bound.

The total overhead of our proposed sparse AllReduce is:

$$2\log_2(P)\alpha + 2\frac{P-1}{P}k\beta \le latency_{total} \le 2\log_2(P)\,\alpha + \left[\left(\frac{\log_2(P)}{2}+1\right)P-1\right]k\beta \quad (6)$$

The pseudo-code of DenseStream representation *top-k* S-SGD is shown in Algorithm 2.

IV. EXPERIMENTS

We conduct our experiments to evaluate our algorithm by real-world applications. We first evaluate the compression efficiency of DenseStream and compare it with the key-value representation under different densities. Then we evaluate the efficiency of DenseStreamAllreduce (DSA) and compare it with the communication algorithms of dense AllReduce (Dense), *top-k* AllGather (TopkA) [5], Dynamic Sparse Allreduce used in SparCML (TopkDA) [5], which are also under different densities. After that, we compare the model training time for one epoch among the four algorithms. Finally, we analyze the convergence of the algorithm.

Note that, the dense AllReduce denotes a single dense AllReduce on a message aggregated from the gradients of all model layers. The data representation of TopkA and TopkDA are key-value.

A. Setup

Our experiments are executed on 4 compute nodes. Each node has an Intel Xeon Gold 5117 CPU with 32GB of RAM, and one NVIDIA RTX2080ti GPU with 12GB global memory. The machines are connected by a 1Gbps Ethernet interface. Machines run 64-bit Ubuntu 18.04 with CUDA toolkit 11.2. The deep learning framework is PyTorch at version 1.7.0. The communication library is mpi4py, which are built with MPICH 3.3.2.

We use VGG16 with 134M parameters and ResNet50 with 24M parameters on cifar-10 dataset. The dataset contains 50000 training samples. We use SGD optimizer with learning rate of 0.01 for VGG16, and 0.001 for ResNet50. the mini-batch size is 64 per machine.

We utilize the *top-k* function provided by PyTorch. the DenseStream sparsification and decompression function programmed by C++ with CUDA, which is accelerated on GPU.

In all experiments, we fix the datatype for storing an index or a length to an **unsigned int**.

B. Compression Efficiency

We use the density of 1%, 5% and 10% to sparsify local gradients for analyzing the compression efficiency of DenseStream. The compression ratio is defined as the ratio of the overhead of DenseStream to that of the key-value representation. The models are trained for 150 epochs, and record compression ratio on each iteration. TABLE I. reports the results. The results are the average of the compression ratio over the entire epoch.

For VGG16, in the case of 1% density, the average compression ratio exceeds 1.0 in epoch 1 and 50. One reason is that the density is extremely low. There are many dense blocks and each block is small, so the additional index and length overhead negate the efficiency. And in the early epoch of training, the training is not stable. We observe that as the training progresses, the compression ratio gradually stabilizes at 1.0.

In the case of 5% density, the average compression ratios are consistent across training stages, the result is 0.87±0.02. DenseStream overhead is less than that of key-value representation. The performance is greatly improved at 10% density. The average compression ratio is 0.75±0.03 in all epochs. Compared with key-value representation, the communication volume is reduced by approximately 1/4.

Surprisingly, for ResNet50, DenseStream performed best in epoch 1 with different densities. The results show that unlike VGG16, the *top-k* gradient value distribution of ResNet50 becomes more uniform as the training progresses. So the performance on ResNet50 is not as good as on VGG16. The best compression ratio is achieved in epoch 1 with the density of 10%, which is 0.82.

The compression efficiency comparison proves the correctness of our hypothesis. During the training of the deep learning model, the *top-k* gradient values show a locality. It also shows that the sparse representation of DenseStream is efficient. Even at extremely low density, it still has good performance. And at high density, it performs much better than key-value representation. When applying DenseStream, zero elements can be replaced by non-*topk* gradient values, which increase the density of local gradients and make model training more robust without increasing storage overhead.

TABLE II. TIME (S) OF COMMUNICATION

Algorithm	VGG16			ResNet50		
	density 1%	*density 5%*	*density 10%*	*density 1%*	*density 5%*	*density 10%*
Dense	7.61			1.34		
TopkA	0.39	1.50	2.91	0.09	0.32	0.59
TopkDA	0.36	1.32	2.59	0.09	0.30	0.51
DSA	0.34	1.01	1.54	0.11	0.26	0.43

C. Communication Speed

Since the three sparse communication algorithms have to sparsify the local gradients before communication and decompress the global gradients after the communication, there are some more steps than dense AllReduce. So we start the measurement at the end of the model backpropagation and finish it after generating the global gradients in a dense format. The communication time includes the time of sparsification, decompression and local gradient accumulation. Meanwhile, in each iteration, the communication overhead of the DenseStream is not fixed, so we average the communication time of the entire epoch 10. The results are presented in TABLE II.

The communication time of the three sparse communication algorithms is much lower than that of dense AllReduce because of the lower communication volume, while the sparsification and decompression overhead are much smaller than communication. In most of cases, our DSA achieves higher efficiency, which is 12.18x faster than dense AllReduce on average. For VGG16, at 1% density, the communication time is reduced by 5% compared to TopkDA. The efficiency improvement mainly comes from the *balanced split* to balance the communication workload between nodes. Meanwhile, the density of resulting global gradients is higher than 1% of local gradients, leading to better compression efficiency and reducing the communication volume in *allgatherv* phase. In addition, we found that the global gradient accumulated from *top-k* local gradient values has higher locality, and the compression efficiency of DenseStream is further improved. This is shown in the experiments of 10% density, compared with other algorithms, the performance of our proposed algorithm has been greatly improved, the communication time is reduced by 43% on average.

For ResNet50, our proposed algorithm performs slightly worse at 1% density, because of the higher sparsification overhead. At 10% density, the communication time is reduced by an average of 21% compared to TopkA and TopkDA.

Overall, our proposed algorithm shows applicability and scalability.

D. A Critical View

We break down the time of communication time into local gradients sparsification time and the real communication time. Then the time of an iteration has three parts: GPU computation time, sparsification time, and communication time. The total time cost of an iteration can be obtained by adding the GPU computation time to the previous results, therefore we focus on the proportion of each part in this section. The results are shown in Fig. 4.

In dense S-SGD, the proportion of computation time is extremely low, and the GPU is idle most of the time. Ignoring the increase in communication latency caused by the increase in the number of computing nodes, nearly a hundred computing nodes can achieve the same throughput as a single node, which is unacceptable.

The results also show a potential problem. There are high sparsification-to-computation ratios. In the training of ResNet50, the overhead of sparsification with 10% density even exceeds the computation overhead, and that with 1% density still exceeds the half computation overhead. The VGG16 has worse performance than ResNet50, which is caused by less computation overhead and more parameters.

Gradient sparsification technique significantly reduces the communication volume, leading to lower communication-to-computation ratio, and improves the performance of data parallel training. When communication is bottleneck, the overhead of gradient sparsification has little impact for distributed training. However, once a higher-speed network is used, such as 10Gig-Ethernet or InfiniBand network, the communication time is reduced by high bandwidth, and the proportion of sparsification time increases in an iteration. Sparsification overhead even become the new bottleneck to restricting the speedup ratio.

The average complexity of quickselect [10] based *top-k* selection is $O(n)$, but not GPU-friendly. The commonly implementation of *top-k* selection on GPU is bitonic *top-k* [2], which has the complexity of $O(n\log^2 k)$, is not good enough for large k. Therefore, the *top-k* sparsification overhead is relatively large. An expensive gradient sparsification algorithm is not properly for data parallel training, even if it has excellent sparsification performance. Our DenseSteam representation algorithm does not rely on *top-k* sparsification, and can be fully accelerated by the GPU, leading to low execution overhead. It can be used with other more efficient sparsification algorithms to reduce the bottleneck caused by sparsification overhead. Another problem with *top-k* sparsification is that, in practice, to reduce the communication time, the gradients exchange is carried out at the neural network layer-wise [21]. The layer-wise exchange allows computation and communication to overlap, reducing GPU idle time. However, *top-k* sparsification can only be executed after the backpropagation of the neural network is completed, and it cannot be applied to the data parallelization of layer-wise gradients exchange. DenseSteam does not have to execute on the overall gradient, and can be used on any tensor, which has better applicability.

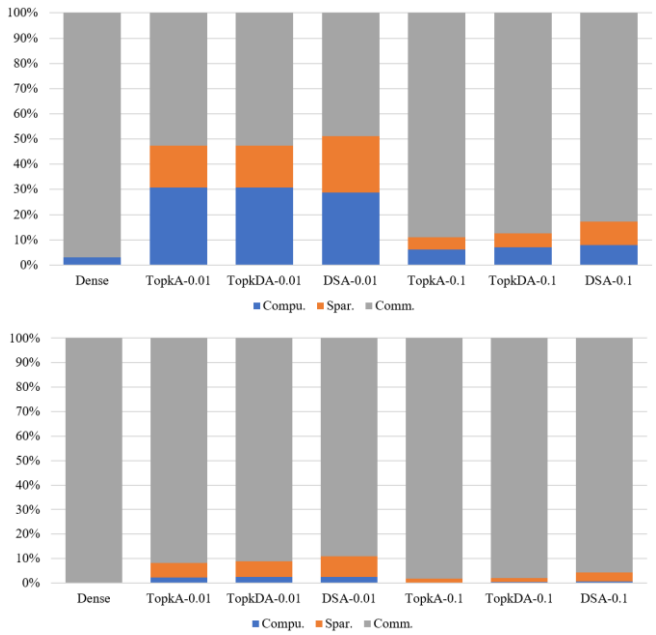

Fig. 4. Time breakdown of computation, sparsification and communication. "Compu." indicates forward and backward computation time, "Spar." indicates sparsification time, and "Comm." indicates the gradients communication time. 0.01 and 0.1 indicates the density of 1% and 10%. **Above**: the result of ResNet50. **Below**: the result of VGG16.

E. Convergence

In much previous work [1][23][26], the convergence of *top-k* S-SGD has been verified. Our algorithm does not modify the content of gradients in *top-k* S-SGD and has no effect on its convergence. We only evaluate the convergence of the VGG16 at 5% density, and the results are shown in Fig. 5. The training loss of the model is almost simultaneous with dense S-SGD at the same epoch, and gradient sparsificatin has only slightly impact on model convergence. But the training time gap of each epoch between dense S-SGD and our DSA S-SGD is huge, and DSA S-SGD can be improved by 12.18x, compared to dense S-SGD. The total model training time also has a large reduction.

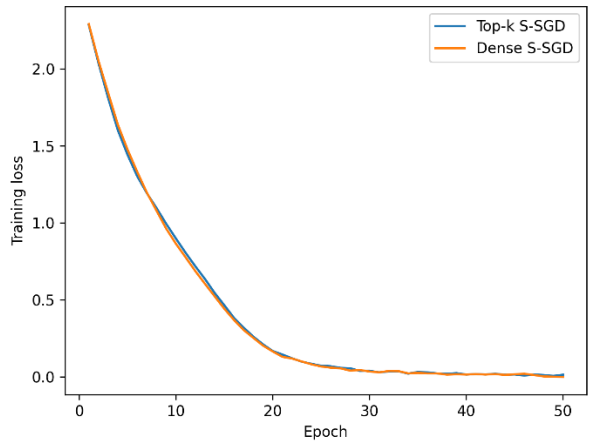

Fig. 5. The convergence of VGG16 on 4 nodes with 5% density.

V. Conclusion

In this paper, we propose a novel data representation for sparsified gradients to alleviate the fill-in problem in data-parallel training. DenseStream enables better performance by utilizing the locality of gradient value statistics. Compared with key-value representation, it is more suitable for neural network training and can achieve higher compression efficiency and larger representation range. The corresponding sparse AllReduce algorithm also has better performance than the counterparts. The experimental results for data-parallel training of real-world neural network models show that DenseStream can provide significantly improvement. In future work, we would like to further investigate more efficient algorithms to reduce the overhead of sparsification, and investigate the sparsification which can be used for other distributed training techniques.

References

[1] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," in The 2017 Conference on Empirical Methods in Natural Language Processing, 2017, pp. 440–445.

[2] A. Shanbhag, H. Pirk, and S. Madden, "Efficient top-k query processing on massively parallel hardware," in Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 1557-1570.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," Advances in neural information processing systems, vol. 33, pp. 1877-1901, 2020.

[4] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan, "Adacomp: Adaptive residual gradient compression for data parallel distributed training," in The 32nd AAAI Conference on Artificial Intelligence, 2018.

[5] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "SparCML: High-performance sparse communication for machine learning," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1-15.

[6] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in Advances in Neural Information Processing Systems, 2018, pp. 5973–5983.

[7] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," arXiv preprint arXiv:1602.06709, 2016.

[8] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," Concurrency and Computation: Practice and Experience, vol. 19, no. 13, pp. 1749-1783, 2007.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770-778.

[10] H. M. Mahmoud, R. Modarres, and R. T. Smythe, "Analysis of quickselect: An algorithm for order statistics," RAIRO-Theoretical Informatics and Applications, vol. 29, no. 4, pp. 255-276, 1995.

[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," The Journal of Machine Learning Research, vol. 18, no. 1, pp. 6869–6898, 2017.

[12] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "SIGNSGD: Compressed optimisation for non-convex problems," in International Conference on Machine Learning, 2018, pp. 559–568.

[13] J. Wu, W. Huang, J. Huang, and T. Zhang, "Error compensated quantized SGD and its applications to large-scale distributed optimization," in International Conference on Machine Learning, 2018, pp. 5325-5333.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," Communications of the ACM, vol. 60, no. 6, pp. 84-90, 2017.

[16] S. Li and T. Hoefler, "Near-optimal sparse allreduce for distributed deep learning," in Proceedings of the 27th ACM SIGPLAN

Symposium on Principles and Practice of Parallel Programming, 2022, pp. 135-149.

[17] L. Wang, W. Wu, J. Zhang, H. Liu, G. Bosilca, M. Herlihy, and R. Fonseca, "FFT-based gradient sparsification for the distributed training of deep neural networks," in Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020, pp. 113-124.

[18] P. Goyal, P. Dollar, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, ´ A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training ImageNet in 1 hour," arXiv preprint arXiv:1706.02677, 2017.

[19] P. Jiang and G. Agrawal, "A linear speedup analysis of distributed deep learning with sparse and quantized communication," in Advances in Neural Information Processing Systems, 2018, pp. 2530–2541.

[20] R. Rabenseifner, "Optimization of collective reduction operations," in International Conference on Computational Science, 2004, pp. 1-9.

[21] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," arXiv preprint arXiv:1802.05799, 2018.

[22] S. Shi, X. Chu, K. C. Cheung, and S. See, "Understanding top-k sparsification in distributed deep learning," arXiv preprint arXiv:1911.08772, 2019.

[23] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified SGD with memory," in Advances in Neural Information Processing Systems, 2018, pp. 4452–4463.

[24] 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.

[25] W. Wang and N. Srebro, "Stochastic nonconvex optimization with large minibatches," in Algorithmic Learning Theory, 2019, pp. 857-882.

[26] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in International Conference on Learning Representations, 2018.