Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks

Soojeong Kim
Seoul National University
soojeong_kim@snu.ac.kr

Gyeong-In Yu
Seoul National University
gyeongin@snu.ac.kr

Hojin Park
Seoul National University
hojinpark.cs@gmail.com

Sungwoo Cho
Seoul National University
sungwoocho@snu.ac.kr

Eunji Jeong
Seoul National University
ejjeong@snu.ac.kr

Hyeonmin Ha
Seoul National University
hyeonmin.ha@snu.ac.kr

Sanha Lee
Seoul National University
sanhaleehana@snu.ac.kr

Joo Seong Jeong
Seoul National University
joosjeong@snu.ac.kr

Byung-Gon Chun*
Seoul National University
bgchun@snu.ac.kr

Abstract

The employment of high-performance servers and GPU accelerators for training deep neural network models have greatly accelerated recent advances in deep learning (DL). DL frameworks, such as TensorFlow, MXNet, and Caffe2, have emerged to assist DL researchers to train their models in a distributed manner. Although current DL frameworks scale well for image classification models, there remain opportunities for scalable distributed training on natural language processing (NLP) models. We found that current frameworks show relatively low scalability on training NLP models due to the lack of consideration to the difference in sparsity of model parameters. In this paper, we propose Parallax, a framework that optimizes data parallel training by utilizing the sparsity of model parameters. Parallax introduces a hybrid approach that combines Parameter Server and AllReduce architectures to optimize the amount of data transfer according to the sparsity. Experiments show that Parallax built atop TensorFlow achieves scalable training throughput on both dense and sparse models while requiring little effort from its users. Parallax achieves up to 2.8x, 6.02x speedup for NLP models than TensorFlow and Horovod with 48 GPUs, respectively. The training speed for the image classification models is equal to Horovod and 1.53x faster than TensorFlow.

*Corresponding author

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303957

CCS Concepts • **Computer systems organization** → **Distributed architectures**; *Neural networks*; *Data flow architectures*; • **Software and its engineering** → *Data flow architectures.*

Keywords sparsity-aware data parallel training, deep learning framework, graph transformation

ACM Reference Format:
Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303957

1 Introduction

It is a common practice nowadays for deep learning (DL) practitioners to utilize a cluster of GPU resources for training deep neural networks. This is mainly motivated by the fact that recent deep neural network architectures involve very large computations [16, 37, 43] and are trained on large datasets [6, 32], typically requiring multiple GPUs in order to finish training within a reasonable time limit. There are a few parallelization strategies for accelerating training on multiple GPUs: running multiple model replicas that process disjoint datasets (data parallelism), partitioning a single model among multiple devices (model parallelism), and a mixture of the previous two strategies (hybrid parallelism). Among these techniques, data parallelism is the most widely used thanks to its simplicity [12, 18, 37], and is supported by most DL frameworks such as TensorFlow [1], PyTorch [29], MXNet [8], Caffe2 [11], and Horovod [34], to increase training throughput by processing data in parallel.

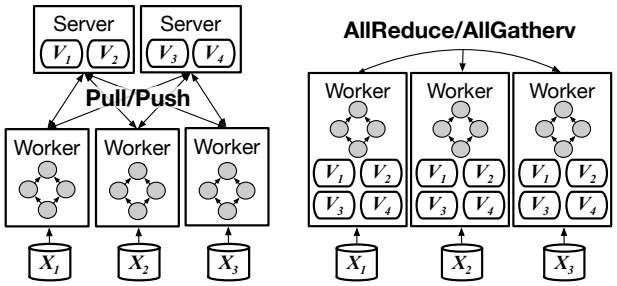
There are a number of recent works that push the limit of data parallel training [2, 12, 17, 20], achieving near-perfect

throughput scaling efficiency[1] of 99.2% with thousands of GPUs [17]. However, all of these works focus on parallelizing image classification models. Little attention has been paid to training models from other domains, namely natural language processing (NLP) models. In fact, we observed that using TensorFlow [1] to train NMT [43] and LM [18] – NLP models for neural machine translation and language modeling, respectively – with 48 GPUs leads to scaling efficiencies of only 19.0% and 7.0% (Section 6). Current solutions to data parallel training are inadequate for handling a certain characteristic of these NLP models: sparsity of model parameters.

Multi-dimensional arrays that hold the parameters of a DL model can be classified into *dense variables* and *sparse variables*[2], depending on how their elements are accessed. For a dense variable, all elements are accessed at least once during a single training iteration. On the other hand, for a sparse variable, only a subset of the elements are accessed in one iteration. Image classification models, such as the Inception-V3 [37] model, usually consist solely of dense variables for convolutional layers and fully connected layers. We refer to such models as *dense models*. In contrast, NLP models have both dense variables and sparse variables. For instance, the aforementioned LM [18] model uses dense variables for internal long short-term memory (LSTM) cell parameters and sparse variables for word embeddings. We define such models as *sparse models*.

Sparse models tend to have larger variables than dense models, and must be dealt with differently in terms of parameter synchronization to maintain reasonable scalability. For example, the largest variable in the dense model Inception-V3, weight of the fully connected layer, has 2.05 million elements, while the largest variable in the sparse model LM, the embedding matrix, has 406 million elements. Synchronizing a large variable across multiple GPUs requires significant network bandwidth and consumes many CPU clocks for aggregating results from GPUs. Thus, naïvely communicating all elements of a large sparse variable, even though only a small subset is accessed, results in relatively low scalability. At the same time, however, treating all variables as sparse variables is inefficient, as there are highly optimized implementations for communicating dense variables across GPUs such as the NCCL [27] library.

In this paper, we introduce Parallax, a framework that takes the sparsity of variables into account to optimize data parallel training. We analyze how the amount of data transfer changes according to whether variables are sparse or dense in two different training architectures: *Parameter Server* and *AllReduce*. Based on this analysis, Parallax pursues a hybrid approach that uses the Parameter Server architecture for handling sparse variables and the AllReduce architecture for handling dense variables. Moreover, Parallax partitions large sparse variables by a near-optimal number of partitions to maximize parallelism while maintaining low computation and communication overhead. Parallax further optimizes training with local aggregation and smart operation placement to mitigate communication overhead. Graph transformation in Parallax automatically applies all of these optimizations and the data parallel training itself at the framework level to minimize user efforts for writing and optimizing a distributed program by composing low-level primitives.



(a) PS Architecture **(b)** AllReduce Architecture

Figure 1. The Parameter Server architecture and the AllReduce architecture.

We have implemented Parallax on top of TensorFlow [1] 1.6 with Horovod [34] 0.11.2. Experiments on two sparse NLP models, LM [18] and NMT [43], and two dense image classification models, ResNet-50 [16] and Inception-V3 [37], show that Parallax can speed up DL training of sparse models while achieving similar performance to state-of-the-art frameworks on dense models. Parallax achieves up to 2.8x and 6.02x speedup for the NLP models compared to TensorFlow and Horovod on 48 GPUs, respectively. The training speed for the image classification models is equal to Horovod and 1.53x faster than TensorFlow. Although we used NLP models for our evaluation to demonstrate the effectiveness of sparsity-aware data parallel training, Parallax's techniques can be applied to any sparse model, such as speech recognition [5, 10] and graph neural networks [19]. Even more, the performance gain is earned with absolutely no manual optimizations from the user – merely a few lines of code are needed to use the Parallax API.

The rest of the paper is organized as follows. Section 2 describes the DL background related to Parallax and the motivation of utilizing model sparsity to optimize distributed training, while Section 3 introduces two sparsity-aware techniques of Parallax. Sections 4 and 5 present the design and implementation of Parallax. Section 6 presents evaluation results. Section 7 presents related work and Section 8 concludes.

[1] Scaling efficiency measures the percentage of speedup (in terms of throughput) in distributed training compared to the ideal, linear speedup when the same amount of GPUs are used.

[2] We use the term *variable*, following TensorFlow. A *sparse/dense variable* is different from a *sparse/dense array*, which has its own mathematical meaning regarding the number of nonzero elements.

2 Background and Motivation

In this section, we briefly discuss data parallel distributed training and its two representative architectures: Parameter Server and AllReduce. We also explain the motivation for taking model sparsity into account when training a DL model in a distributed manner.

2.1 Data Parallel Distributed Training

A DL model refers to a neural network architecture, which is trained via gradient descent; the loss value of the model is calculated from forward computations, and the loss is passed back through the model according to the backpropagation algorithm to compute gradients. These gradients are then used to update corresponding variables that compose the neural network. Data parallel distributed training is utilized to process several mini-batches simultaneously with multiple GPUs. GPUs are set to perform the same computation on different mini-batches, each producing a unique set of gradients. In case of asynchronous training, the gradients from one GPU are used to update variables without waiting for other GPUs. On the other hand, for synchronous training, all GPUs wait for one another to finish their gradient computation for variables. Then, the computed gradients are aggregated before being used to update corresponding variables. For both asynchronous and synchronous training, data communication between GPUs and machines is necessary to share the computed gradients.

For asynchronous training, the staleness of model variable updates is known to negatively impact the model's accuracy and produce relatively unpredictable results [7, 13, 45]. Thus, many DL models are trained synchronously [12, 28, 36, 43]. This paper also assumes synchronous training, although we note that Parallax supports both synchronous and asynchronous training.

Data Parallel Training Architectures Two widely-used data parallel distributed training architectures are the Parameter Server (PS) [21] architecture and the AllReduce (AR) architecture. The PS architecture, initially proposed for topic modeling [21], has been extensively used in previous works [1, 8, 9] thanks to the scalable structure that allows a large set of variables to be distributed into multiple machines. A typical PS architecture consists of *server* and *worker* processes as described in Figure 1(a). Server processes store subsets of model variables ($V_1, \ldots, V_4$) in memory, while worker processes *pull* variables from servers to perform local computations on their respective mini-batches (X_1, X_2, X_3) and later *push* gradients with respect to variables back to servers. As a result, variable synchronization between workers is done indirectly via server processes.

For the AR architecture, there is no process dedicated just for holding variables, as shown in Figure 1(b). Rather, all workers are given a replica of variables and share locally computed gradients via collective communication primitives such as `AllReduce` [25, 30] and `AllGatherv` [39]. `AllReduce` reduces values from all processes to a single value, while `AllGatherv` simply gathers the values from all processes. More formally, for the gradient $\frac{\partial L}{\partial v}(X_i)$ of a loss function L with respect to a variable v given a mini-batch data X_i, where worker i processes X_i ($i \in 1, \ldots, N$), `AllReduce` aggregates gradients from all workers by computing the sum of gradients $\sum_{i=1}^{N} \frac{\partial L}{\partial v}(X_i)$. On the other hand, `AllGatherv` aggregates gradients by concatenating the gradients into $[\frac{\partial L}{\partial v}(X_1), \ldots, \frac{\partial L}{\partial v}(X_N)]$. Then, these primitives broadcast the aggregated gradients back to all processes. The replica of variables housed in each worker is updated using the aggregated gradients, thereby all replicas in different workers are always synchronized. This collective mechanism makes data parallel training simple because all workers always have the same variable values, thus there are no synchronization issues regarding variable updates. Since the AR architecture is easier to use and shows better performance compared to the PS architecture for image classification models [34, 35], recent attempts to scale out DL training [2, 12, 17, 20] employ AR as their distributed training architecture.

A major collective communication implementation used for the AR architecture is NCCL [27], a well-known collective communication library that takes advantage of the GPU topology within and across multiple machines. Depending on how GPUs are connected in a machine and across machines, NCCL composes different ring structures to achieve better performance. It provides a highly optimized communication implementation, which is especially effective when the GPUs in the cluster support GPUDirect P2P or GPUDirect RDMA [26]. Most DL frameworks that support distributed training, such as TensorFlow [1], PyTorch [29], MXNet [8], Caffe2 [11] and Chainer [38], adopt NCCL as their collective communication implementation.

2.2 Necessity of Sparsity-awareness

Although existing DL frameworks demonstrate scalable performance for data parallel training on large GPU clusters, their results are mostly based on well-known image classification models; there still remain untapped opportunities for scaling distributed training for models with sparse variables. A representative example of a sparse variable would be an embedding matrix, which maps a word to an embedding vector. Since sentences in a mini-batch typically include only a subset of an entire vocabulary list, only the corresponding rows of the embedding matrix is read and updated at each iteration. For efficient memory management and computation, most DL frameworks provide special data structures for handling sparsity. Instead of using a single array to represent sparse data such as a gradient of a sparse variable, two separate arrays are used – one for the actual values, and another for indicating the value indices within the data, similar to the compressed sparse row (CSR) format [33]. For

Models	# Elements		α_{model}	Throughput	
	Dense	Sparse		PS	AR
ResNet-50	23.8M	0	1	5.8k	**7.6k**
Inception-v3	25.6M	0	1	3.8k	**5.9k**
LM	9.4M	813.3M	0.02	**98.9k**	45.5k
NMT	94.1M	74.9M	0.65	**102k**	68.3k

Table 1. The total size of dense and sparse variables, α_{model}, and the training throughput (images or words per sec) of PS and AR architectures for four DL models, including two image classification models (ResNet-50, Inception-v3) and two NLP models (LM, NMT). The experiments are conducted on 48 GPUs using the cluster environment, datasets, and batch sizes described in Section 6. The PS column shows the results of TensorFlow using the PS architecture, and the AR column shows the results of Horovod using `AllReduce` for dense variables and `AllGatherv` for sparse variables.

example, TensorFlow [1] manages dense data using a `Tensor` abstraction, while sparse data correspond to `IndexedSlices` or `SparseTensor` that contain two `Tensors` to hold nonzero indices and values separately.

We claim that just like the data structures for sparse data, distributed data parallel training should also be aware of the different characteristics of dense and sparse variables. To support this statement, we conducted experiments to show how the performance trend of training sparse models differs from that of dense models, regarding the underlying training architecture as well as partitioning variables for the appropriate architecture. Moreover, this claim is further backed by researches from the machine learning community that employed data parallel training to train sparse models [18, 43].

Choosing Appropriate Training Architectures Table 1 shows that the sparsity of a model is an important factor when selecting a distributed training architecture. It depicts the training throughput of four DL models along with their variable sizes and a ratio factor α_{model} that describes how sparse the model parameters are. α_{model} is a weighted sum of α values of variables in the model, where the weight of each variable is proportional to its number of elements. We define the α value of a variable as the average ratio of the number of elements that are actually used by a worker in one iteration to the total number of elements. The first two models in the table, ResNet-50 and Inception-v3, are dense models and thus they do not contain sparse variables. The next two models, LM and NMT, are sparse models, containing both dense and sparse variables.

Results show that the AR architecture is preferable for dense models, while the PS architecture performs better for sparse models. This is because different distributed training architectures use network bandwidth in different ways; we discuss this further in Section 3.1. To the best of our knowledge, no prior work considers the sparsity of models when selecting the distributed training architecture.

Model	# Partitions					
	8	16	32	64	128	256
LM	50.5k	78.6k	96.5k	96.1k	98.9k	93.2k
NMT	90.7k	97.0k	96.5k	101.6k	98.5k	100.0k

Table 2. Training throughput (words/sec) according to the number of partitions for LM and NMT models, using the PS architecture. The experiment setup is the same as Table 1.

Impact of Partitioning Sparse Variables When using the PS architecture, it is common to partition large variables into multiple pieces to overcome memory constraints or to reduce load imbalance between server processes. However, even when the memory requirements are satisfied and there is no significant load imbalance present, the number of partitions of sparse variables can affect overall performance.

Table 2 shows the throughput of training the sparse models, LM and NMT, on various numbers of sparse variable partitions using the PS architecture. Although all cases satisfy memory constraints and avoid significant load imbalance, the performance improvement for using the best possible choices (128 and 64 partitions for LM and NMT, respectively) and the worst possible choices (8 partitions for both models) is meaningful for both models; 1.98x for LM and 1.12x for NMT. It is also worth noting that blindly increasing the number of partitions is not optimal, as the throughput at 256 partitions is worse than at 128 partitions in the LM model. The performance improvement comes from the parallelization of operations for sparse variables; we describe the reasons for speedup in detail in Section 3.2.

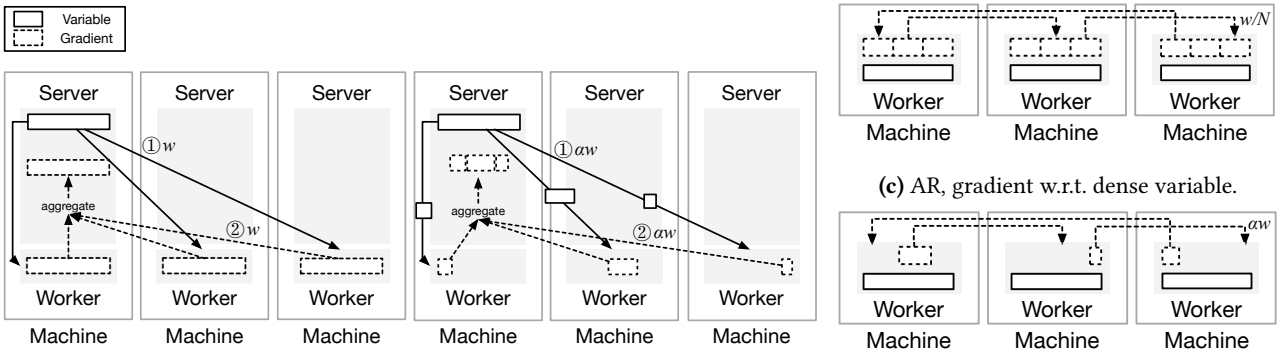
3 Sparsity-aware Data Parallel Training

Motivated by the experiment results in Section 2.2, we propose two sparsity-aware techniques to improve the performance of distributed training for sparse models: 1) a hybrid architecture of PS and AR, and 2) automatic searching of the optimal number of sparse variable partitions.

3.1 Hybrid Architecture

As shown in Table 1, the training speeds of the PS and AR architectures are affected by model sparsity. The PS architecture performs faster when the model is sparse, while the AR architecture shows better performance when the model is dense. We analyze this trend further by formulating the size of data transferred across the network during one training iteration for both architectures.

Figure 2 shows how each training architecture synchronizes progress from multiple workers, for dense and sparse



(a) PS, dense variable and its gradient. **(b)** PS, sparse variable and its gradient. **(c)** AR, gradient w.r.t. dense variable. **(d)** AR, gradient w.r.t. sparse variable.

Figure 2. Data transfer for each type of variable and its gradient from/to a machine according to the training architecture. Bars with solid edge represent variables, while bars with dotted line represent gradients. Gradients of a dense variable from different workers are reduced by computing a sum of the gradients, while gradients of a sparse variable are aggregated by concatenating the arrays.

variables. To simplify the explanation, we assume each machine contains only one worker process, and a server process is colocated with the worker in the PS architecture case. Moreover, w refers to the average size of variables as bytes, N is the number of machines, and α is the element ratio of variables defined in Section 2.2.

Regarding a single dense variable in the PS architecture, a server process sends w bytes to $N-1$ machines each, resulting in a network transfer of $w(N-1)$ bytes (Figure 2(a)①). The network cost for a server process occurs only for $N-1$ machines instead of all N machines because server and worker processes in the same machine communicate locally within the machine without involving network communication. Similarly, the server receives gradients of the same size back from the $N-1$ machines, leading to another $w(N-1)$ bytes (Figure 2(a)②). Thus, for a single dense variable, the machine that houses the corresponding server sends and receives a total of $2w(N-1)$ bytes of data over the network for each iteration. The network transfer for a sparse variable (Figure 2(b)) is similar to the dense variable case; as defined in Section 2.2, each worker utilizes α of the elements in a sparse variable (in average), so they fetch αw bytes from the server and push back the same amount of gradients to the server.[3] Therefore, the amount of network transfer for a sparse variable becomes $2\alpha w(N-1)$.

The data transfer behavior of the AR architecture varies depending on the actual algorithm implementation of `AllReduce` and `AllGatherv` being used. Here, we assume the ring algorithm [31], one of the most popular collective communication algorithms, which is used in the NCCL library. Regarding a single dense variable, each worker sends and receives w/N bytes of data for $2(N-1)$ communication steps, where gradients are reduced for the first $N-1$ steps and the reduced values are broadcast back to all workers for the next $N-1$ steps. Figure 2(c) shows the algorithm for one communication step, in which $2w/N$ bytes are going into and out of a single worker via network transfer. Repeating this for $2(N-1)$ steps, we get a grand total of $4w(N-1)/N$ bytes for a machine. On the other hand, for a sparse variable, each worker sends and receives αw bytes of data for $N-1$ communication steps in order to `AllGatherv` gradients for that variable (Figure 2(d)), resulting in $2\alpha w(N-1)$ bytes of network transfer for each machine. The One Variable column of Table 3 summarizes these discussions about network transfer for a single variable, depicting all possible combinations of dense or sparse variables and the PS or AR architectures.

Type	Arch	One Variable	m Variables
Dense	PS	$2w(N-1)$	$4wm\frac{N-1}{N}$
	AR	$4w\frac{N-1}{N}$	$4wm\frac{N-1}{N}$
Sparse	PS	$2\alpha w(N-1)$	$4\alpha wm\frac{N-1}{N}$
	AR	$2\alpha w(N-1)$	$2\alpha wm(N-1)$

Table 3. The amount of network transfer required per machine for each type of variable according to the training architecture.

Moving from one variable to multiple variables, we add additional assumptions about the variable distribution across servers. We assume that all variables occupy the same amount of memory (w bytes) and are distributed evenly across server processes. In such a balanced PS architecture, each machine manages $\frac{m}{N}$ dense variables where m is the number of dense variables in a model. For the $\frac{m}{N}$ variables that a machine manages, a total of $2w(N-1)\times\frac{m}{N}$ bytes of network transfer

[3] We omitted the network transfer for exchanging nonzero indices since it is negligible in most cases compared to nonzero values.

occurs; for the other $m - \frac{m}{N}$ variables that the machine does not manage, $2w \times (m - \frac{m}{N})$ bytes of transfer occur since the machine needs to fetch w bytes for each variable and send another w bytes for each corresponding gradient. Thus, the amount of network transfer per machine for m dense variables becomes $2w(N-1) \times \frac{m}{N} + 2w \times (m - \frac{m}{N}) = 4wm\frac{N-1}{N}$. Similarly, each machine requires $4\alpha wm\frac{N-1}{N}$ bytes of network transfer in order to synchronize m sparse variables.

Unlike the PS architecture, all variables in the AR architecture are housed by all workers, and thus are present in all machines. Thus, we can simply derive the total amount of network transfer per machine by multiplying m with the amount of network transfer for a single dense or sparse variable, giving us $4wm\frac{N-1}{N}$ and $2\alpha wm(N-1)$ bytes, respectively.

Both PS and AR architectures require the same amount of network transfer for a machine, with m dense variables. However, the amount required for a single dense variable that is managed by the machine is much larger in the PS architecture. The machine that manages the variable needs to handle $2w(N-1)$ bytes of network transfer, compared to $2w$ bytes of other machines. This difference can possibly lead to a communication bottleneck in the machine in charge of the variable, while network bandwidth for other machines is under-utilized. We anticipate this asymmetry between machines to be the root cause of the performance difference between PS and AR architectures. Since a DL model comprises multiple layers and there are dependencies between them, pull and push requests for variables in different layers are scattered along the timeline. On the other hand, there is no such asymmetric network transfer for the AR architecture, and therefore no particular machine becomes a bottleneck. Recent studies [34, 35] show that the NCCL-based AR architecture achieves higher performance on dense models such as ResNet-50 [16], Inception-v3 [37], and VGG-16 [36].

For sparse variables, exchanging gradients using the AR architecture requires much more data transfer compared to the PS architecture. As N becomes larger, the difference between the two architectures becomes more significant.

Based on the analysis, we propose a hybridization of the two architectures to achieve the best of both worlds. Parallax employs a hybrid architecture in which the AR architecture handles dense variables and the PS architecture handles sparse variables. Each worker has a replica of dense variables, while separate server processes manage only sparse variables. Note that if the α value of a sparse variable is close to 1, then it may be helpful to handle the variable as a dense variable and use `AllReduce`, even though it requires $\frac{1}{\alpha}$ times larger network transfer compared to the PS architecture. In this case, α should be large enough to make the gain from efficient network utilization of the AR architecture surpass the overhead of extra network transfer.

3.2 Partitioning of Sparse Variables

As stated in Section 2.2 and Table 2, partitioning sparse variables can affect training throughput. The fact that the performance goes up as the number of partitions increases up to 128, without any significant load imbalance, implies that there is inevitably another factor that contributes to the improvement.

We found that partitioning sparse variables effectively parallelizes the aggregation of the corresponding gradients, as well as the variable update operations. Gradient aggregation and update operations for sparse variables require iterating through nonzero indices one by one to accumulate values with the same index. Partitioning a sparse variable parallelizes these operations by dividing incoming values and indices into disjoint sets, and thus enables the parallel execution of such operations. Meanwhile, increasing the number of partitions introduces additional overhead for stitching the partial results from each partition into one tensor to be used as input for other operations [1]. It is also accompanied with the overhead of managing each partition of the variable as separate arrays. These aspects are related to not only the DL model itself, but also the hardware specification of the cluster and batch size; simple rule-based heuristics are not able to find a reasonable optimum for various conditions.

To capture these effects, we suggest a cost-based model that predicts iteration time as a function of the number of partitions P:

$$iter\_time = \theta_0 + \theta_1 * \frac{1}{P} + \theta_2 * P \quad (1)$$

Parameter θ_0 represents the constant cost for fixed computation and communication, which does not change over P. θ_1 captures the cost that can be parallelized and amortized by increasing P, while θ_2 represents the overhead incurred by partitioning sparse variables.

Parallax collects data points required to fit Equation 1 by performing actual training with different values for P, for a few iterations.[4] Then, we fit the equation using mean-squared error of the sampled iteration time and prediction. In order to reduce the number of samples while maintaining high accuracy, Parallax exploits the fact that Equation 1 is a convex function of P. Setting P's initial sample point to be the number of machines, Parallax collects the iteration time for P while doubling the value until the iteration time starts to increase. Next, Parallax repeats the process while halving P, again until the iteration time starts to go up. The critical point of the convex function is located between the minimum and maximum Ps of the collected data, hence the cost model can predict the optimal P without performing any extrapolation.

[4] Parallax runs 100 iterations and discards values from the first 50 iterations to eliminate startup cost.

```
1  import parallax
2
3  # create a graph as distributed version
4  with single_gpu_graph:
5    ds = input_files_dataset()
6    ds = parallax.shard(ds)
7    en_texts, de_texts = ds.get_input_data()
8
9    with parallax.partitioner():
10     emb_enc = get_variable(shape=[...])
11     emb_dec = get_variable(shape=[...])
12   loss = build_NMT_model(en_texts, de_texts,
13                          emb_enc, emb_dec)
14   grads_and_vars = compute_grads(loss)
15
16   opt = GradientDescentOptimizer(LR=0.1)
17   train_op = opt.update(grads_and_vars)
18
19 graph_runner = parallax.get_runner(
20   single_gpu_graph,
21   resource_info_file,
22   parallax_config)
23
24 for i in range(num_iters):
25   graph_runner.run(train_op)
```

Figure 3. Example code for training the NMT model in a distributed multi-GPU environment with Parallax. Red lines represent the necessary modifications for adding Parallax: `shard` for splitting the input data for data parallelism, `partitioner` for partitioning sparse variables, and `get_runner` for performing automatic parallelization.

4 System Design

Parallax is a sparsity-aware data parallelization framework built on TensorFlow [1], a state-of-the-art DL framework. Parallax enables users to utilize distributed multi-GPU environments when they have a single-GPU computation graph (i.e., a deep learning model developed for training on a single GPU). It guarantees transparency while keeping scalable performance using a hybrid architecture with optimally partitioned sparse variables. For the transparency, users do not need to write new code for data parallel training that requires prior knowledge for training architectures and sparsity of variables. Instead, the framework provides an API that receives a single-GPU computation graph as input and automatically transforms the graph into a multi-GPU, multi-machine computation graph.

4.1 Programming Interface

Parallax provides simple programming interfaces: `shard`, `partitioner`, and `get_runner`. Unlike single-GPU training, input data must be divided into disjoint subsets to be processed by different GPUs for data parallel distributed training. Parallax helps this process with the `shard` API, which receives input data and splits the data into multiple subsets so that each GPU can read a unique subset. When exploration for optimal partitioning is required through `partitioner`, the variables within `partitioner` context are partitioned using an optimal number of partitions searched by Parallax. `get_runner` is the main interface that accepts a single-GPU graph as well as resource information including the IP addresses (or hostnames) of machines and GPU IDs, and an optional Parallax configuration (`ParallaxConfig`) object specifying extra arguments if needed. The configuration includes whether to use local aggregation or not, a file path to save trained variables and aggregation methods for each type of variable indicating whether to compute the average of gradients for dense (or sparse) variables over all GPUs or to compute the sum instead.

We illustrate how to use the Parallax API with a code snippet example for training the NMT [43] model, a DL model for language translation. Figure 3 shows code for training the NMT model on a GPU cluster. Parallax requires three modifications compared to a corresponding single-GPU training code: splitting input data across GPUs (line 6), creating partitioned variables using `partitioner` (line 9), and creating Parallax's `graph_runner` instead of the original framework's. First, a graph object is declared, `single_gpu_graph`, which is followed by the logic for preprocessing input data, the loss function, the gradients from backpropagation, and the gradient descent method for updating the variables (lines 4-17). The input data must be split across GPUs for data parallelism, and this can be accomplished with the `shard` interface. The `ds` object in line 5 represents the whole input data, while the `ds` object returned by `shard` in line 6 is a unique subset of dataset for a model replica. Next, users can create partitioned variables using `partitioner` in line 9. Parallax finds and applies the optimal partitioning for the variables (`emb_enc` and `emb_dec`). Note that each `partitioner` partitions variables into the same number of partitions. When the user wants to partition variables in different granularities, multiple `partitioners` must be created and applied independently. Then, the computation graph is transformed to be executable on multiple GPUs through the `get_runner` interface. In lines 19-22 and line 25, the `graph_runner` object returned by the `get_runner` interface should be used in place of the graph runner of the original framework, since it is not aware of the fact that the computation graph has been converted for a distributed multi-GPU environment.

In the existing frameworks [1, 8], users must use different APIs for constructing computation graphs depending on whether the training is done on a distributed environment or only on a single GPU. For example, a user that wants to train a model using TensorFlow's PS architecture must be aware of two types of processes - server and worker - and

insert mechanisms for gradient aggregation and synchronization. Meanwhile, Parallax lets users recycle almost the same single-GPU code for constructing computation graphs on distributed environments, allowing easier utilization of multiple GPUs. We discuss this point further in Section 7.

4.2 Execution Model

We outline the overall execution model of Parallax as follows. After a client initiates a job with a computation graph and resource information, Parallax analyzes the computation graph to construct hybrid architecture. If the graph only contains dense variables, Parallax launches workers as many as the number of GPUs. On the other hand, if sparse variables are included in the graph, Parallax launches a server process for each machine and a worker process for each GPU. When the processes are launched, the number of partitions for sampling is passed to the workers. Worker processes transform the input graph to a distribute version and run for a small number of iterations on the given resources. During the graph transformation step, Parallax separates dense and sparse variables and creates a distributed graph for AR and PS architectures (if necessary). Then, each worker sends its execution time to the master process which collects execution time according to the number of partitions. This process is repeated until sampling for variable partitioning ends. Finally, Parallax executes the transformed graph with optimally partitioned sparse variables. Next, we explain the details of graph transformation.

4.3 Automatic Graph Transformation

Parallax carries out the transformation process adhering to several specific rules systemically as a substitution of user's manual modifications from a single-GPU graph to a distributed version. Parallax builds transformation rules for AR and PS architectures while maintaining transparency, correctness and scalability, and these rules are combined for hybrid architecture. Note that the transformation rules do not automate hyperparameter tuning to find optimal hyperparameters such as learning rate or batch size. Parallax uses hyperparameters that are given from the input graph.

Transformation for AR Figure 4 shows graph transformation for AR architecture. It is relatively straightforward compared to the transformation for PS because each device carries individual copies of global states (i.e., variables) and does not access states on the other devices. Parallax replicates all operations in the original single GPU graph and places a replica for each GPU in the resource specification. The transformation is simple because of the homogeneity of all the processes (workers) that participate in training, unlike the PS architecture. Parallax automatically identifies gradients using information in a single-GPU graph to satisfy a transparent graph transformation. To aggregate gradients across devices, `AllReduce` operations take place between

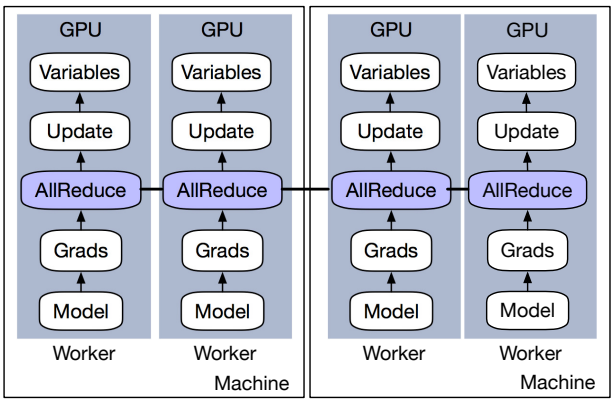


Figure 4. Graph transformation for AR architecture. In this example, the transformed graph uses `AllReduce` to aggregate gradients.

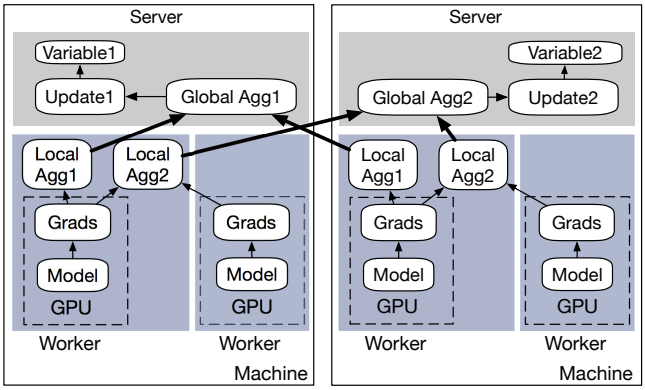


Figure 5. Graph transformation for PS architecture.

operations that produce gradients using backpropagation (`Grads`) and their successors (`Models`).

Transformation for PS Parallax supports an optimized PS architecture using local aggregation and assigning operations effectively across machines. Consequently, the graph transformation rules for PS are defined based on the optimized PS. Parallax transforms a single-GPU graph for PS architecture by creating a copy of forward and backward graph operations for each worker and distributing variables and their update operations across servers. Parallax applies different replication and operation placement policies to variables, variable update operations, and main computation operations. Figure 5 shows an example of the graph transformation. Parallax launches a (parameter) server on each machine and a worker on each GPU in the given resource specification. This colocation of workers and a server in a machine works well since workers are GPU-intensive while servers run lightweight computation, which runs only on CPUs. Parallax evenly distributes variables (`VariableN`) across servers, and a large variable is partitioned to multiple pieces if the variable is specified as a partitioning target in

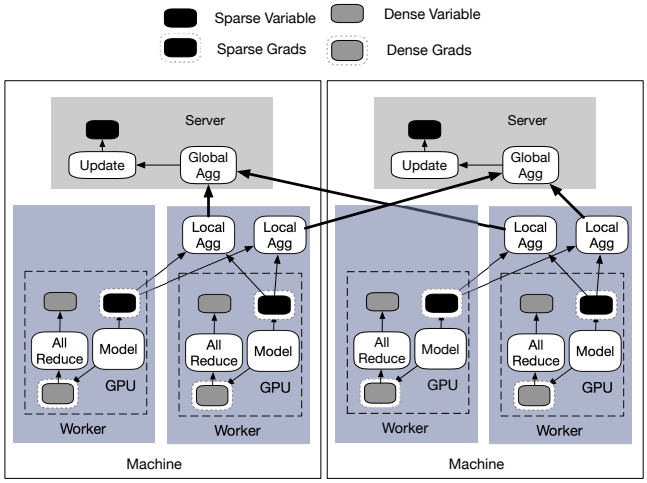


Figure 6. Graph transformation for hybrid architecture.

the code. Each partitioned piece has a partitioned gradients aggregation and a partitioned update operation. Parallax assigns update operations (`UpdateN`) in the same server with their variables to be updated. Identifying model variables and their update operations is feasible because DL frameworks [1, 4, 8, 11] treat them differently from mathematical operations, such as add or multiplication. Main computation operations that are used to compute gradients are replicated as many as the number of GPUs. `Model` and `Grads` represent operations for forward computation and backpropagation, respectively. Along with the detection of gradients, Parallax identifies main computation operations by searching all the ancestor operations from the gradients in the graph. Gradients from each GPU are aggregated twice using aggregation operations for GPUs within a machine (`LocalAggN`) and between machines (`GlobalAggN`). The local aggregation reduces the amount of data communication between workers and servers, which is more expensive than communication between GPUs in the same machine. The outputs of `GlobalAggN` are used to update model variables. Parallax places a global aggregation operation (e.g., `GlobalAgg1`) on the same server with the variable (e.g., `Variable1`) to minimize data transfer between machines.

Transformation for Hybrid Figure 6 shows the transformed graph for hybrid architecture. Regardless of architectures, main computations (`Models` and `Grads`) are replicated in each GPU. Then, Parallax separates dense and sparse variables using the different data structures to handle gradients of each type. Finally, a sparse variable follows PS transformation rules while AR transformation rules are applied to a dense variable. The sparse variables are shared via server processes and global aggregation methods are inserted between locally aggregated gradients from each machine and update operations. The dense variables replicated in each worker are updated using the aggregated gradients from `AllReduce`. Because each variable is synchronized independently, applying different rules to each type of variables completes graph transformation for hybrid architecture.

5 Implementation

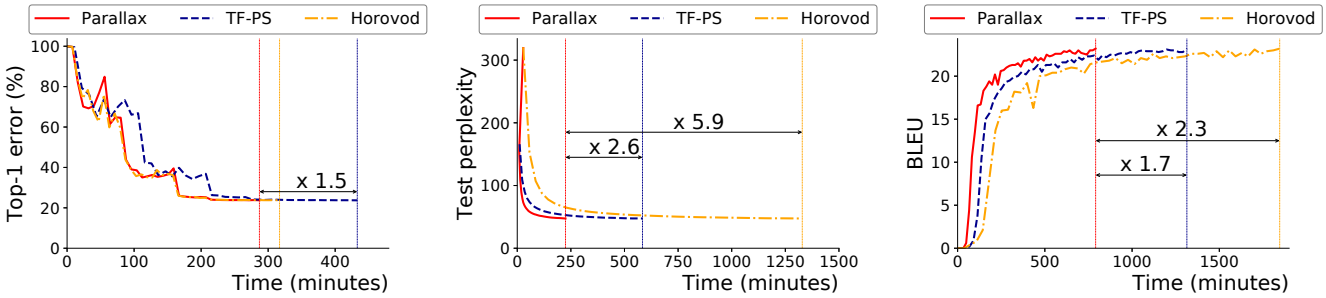
We implemented Parallax on TensorFlow [1] v1.6 with `AllReduce` operation using NCCL in Horovod [34] v0.11.2. We implemented the graph transformation and distributed execution in Python.

Identifying the sparsity of a variable In TensorFlow, dense and sparse variable are distinguished by the different types of their gradient tensors. The type is determined when the gradient tensor is generated by automatic differentiation, depending on how the variable is used in the forward computation. For example, TensorFlow creates a sparse type gradient tensor for a variable used in a sparse access operation, `gather`. Parallax uses this type information to identify if a variable is either sparse or dense.

Graph transformation Graph transformation of Parallax consists of inserting gradient aggregation operations for sparse variables and placing operations to specific resources. Placing operations can be done with the `tf.device` API. However, aggregating gradients requires additional steps as follows. We first place accumulators on servers to aggregate the gradients of sparse variables, where each accumulator handles gradients of a single sparse variable. When gradients are aggregated in an accumulator, a worker asks the server to read the aggregated gradient from the accumulator and update the corresponding variable.

To provide correct variable updates as done in a single-GPU code, Parallax ensures that only one worker, namely a chief worker, triggers the operations for reading aggregated gradients and updating variables. The other workers wait until these variable update operations are finished. The chief's notification arrives through shared queues on each worker. If the other workers also need aggregated gradients to trace their status during training or to compute a global norm of gradients for clipping, Parallax changes the worker-side graphs to read the aggregated gradients from the variables where the chief worker saves them temporarily after reading from accumulators. In case of local aggregation, Parallax adds additional accumulators to each machine, and a worker in the machine becomes a local chief worker to collect gradients within a machine and send them to servers.

In addition, we modified the TensorFlow core to store gradients information, which is the result of auto-differentiation for model variables, in `MetaGraphDef` protobuf in TensorFlow. The modified `MetaGraphDef` enables Parallax to track exact mapping between model variables and their gradients. Parallax uses this information for inserting gradient aggregation operations.



(a) ResNet-50 **(b)** LM **(c)** NMT

Figure 7. Convergence results. (a) Top-1 validation error of ResNet-50. (b) Test perplexity of LM. (c) BLEU score of NMT. The vertical lines represent where each framework reaches the same target value for Top-1 validation error, test perplexity, or BLEU score. The target values are 23.74% for ResNet-50, 47.5 for LM, and 22.5 for NMT.

6 Evaluation

We evaluate Parallax with experiments to answer the following questions:

- Does Parallax correctly transform computation graphs and improve convergence speed using the sparsity-aware data parallel training? (Section 6.2)
- Does Parallax scale out well to multiple GPUs and machines? (Section 6.3)
- How much performance benefits do Parallax's optimization techniques provide? (Sections 6.3, 6.4 and 6.5)
- How does Parallax's performance change under various sparsity degrees? (Section 6.6)

6.1 Experiment Setup

Cluster Configuration. We conducted all the experiments on a GPU cluster of 8 machines. Each machine is equipped with two 18-core Intel Xeon E5-2695 @ 2.10 GHz processors with 256 GB RAM and 6 NVIDIA GeForce TITAN Xp GPU cards. The machines are connected via Mellanox ConnectX-4 cards with 100Gbps InfiniBand. They run Ubuntu 16.04, CUDA 9.0, cuDNN 7, OpenMPI v3.0.0, and NCCL v2.1.

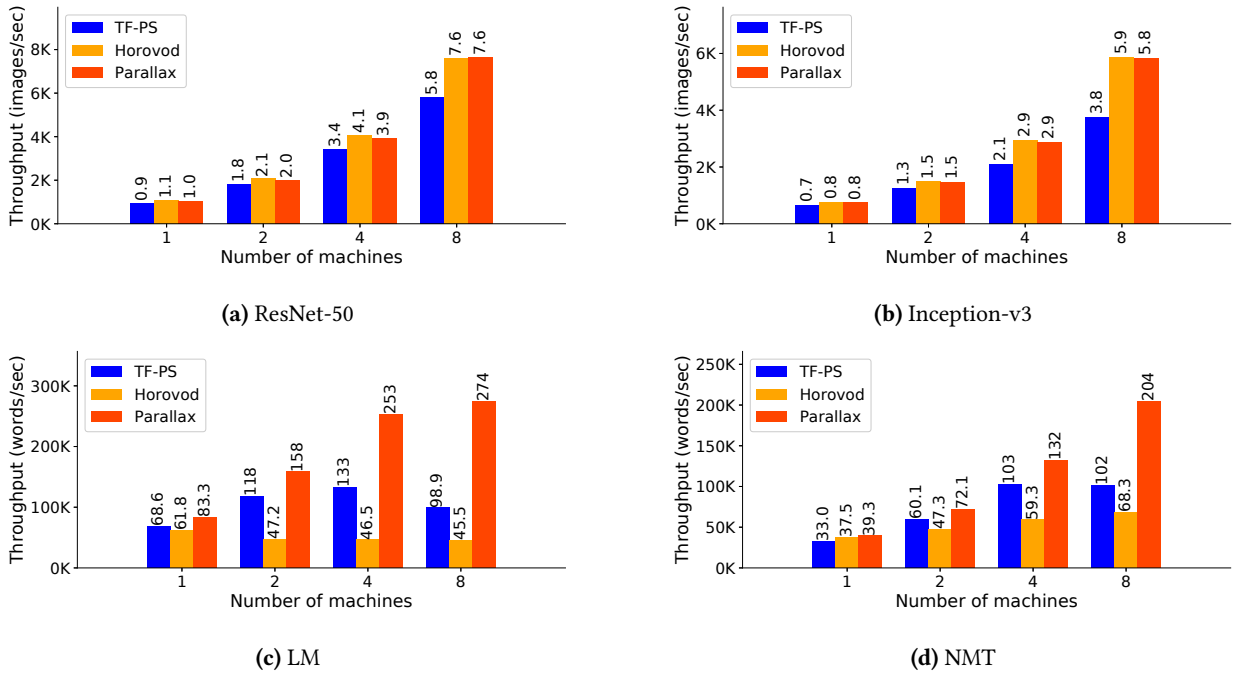
Frameworks. As baselines, we selected TensorFlow v1.6 as a representative DL framework for the PS architecture, and Horovod [34] v0.11.2 on TensorFlow for the AR architecture. In the evaluation, TF-PS denotes TensorFlow with PS. We let Horovod use NCCL for `AllReduce` since NCCL provides highly-optimized communication between GPUs compared to OpenMPI. However, we inevitably use OpenMPI for `AllGatherv`, which is not provided by NCCL.

Models and Datasets. We trained two image classification models and two NLP models in our experiments. ResNet-50 [16] and Inception-v3 [37], are trained with the ImageNet (ILSVRC 2012) [32] dataset that has 1.28M training images and 50K validation images in 1000 categories. LM [18] is a language model that learns a probability distribution over sequences of words in a language. It consists of a single layer of LSTM with hidden state of size 2048, projected to a 512-dimensional embedding. We trained the LM model on the One Billion Word Benchmark [6] that contains one billion words with the vocabulary size of 800K. NMT [43] is a machine translation model, composed of 8-layer LSTMs of 1024 units with a bidirectional encoder of 1024-dimensional embedding. We used the WMT English-German dataset [40] that has 4.5M sentence pairs for NMT model training. As described in Table 1, the image models are dense models, which consist of only dense variables, while the NLP models are sparse models, which contain both dense and sparse variables. The batch size per GPU is 64 for ResNet-50 and Inception-v3, and it is 128 for LM and NMT.

6.2 Model Convergence

Parallax correctly converges models as other frameworks, and the convergence speed is faster than or equal to TF-PS and Horovod. Figure 7 shows the convergence graphs of ResNet-50, LM, and NMT models. We compare the training time taken for each framework to converge models, which is indicated by a model-specific metric reaching the same target values. The target values are 23.74% top-1 error for ResNet-50 experiments (Figure 7(a)), perplexity of 47.5 for LM experiments (Figure 7(b)), and BLEU score of 23.2 for NMT experiments (Figure 7(c)). ResNet-50, LM, and NMT experiments use 48, 36, and 24 GPUs, respectively.

The convergence speed in Figure 7 demonstrates the relationship between the training architecture and the sparsity of models. For example, ResNet-50 results confirm our findings that the AR architecture (Horovod) is efficient for the training of dense models than the PS architecture (TF-PS). Horovod's training takes less time than TF-PS for the same top-1 validation error. Parallax shows almost equal performance with Horovod because Parallax utilizes only the AR architecture for dense models by using Horovod `AllReduce`



(a) ResNet-50 **(b)** Inception-v3

(c) LM **(d)** NMT

Figure 8. Training throughputs of (a) ResNet-50, (b) Inception-v3, (c) LM and (d) NMT on Parallax, TF-PS and Horovod, varying the number of machines from 1 to 8. In dense models (ResNet-50 and Inception-v3), Parallax outperforms TF-PS and shows the same performance with Horovod. In sparse models (LM and NMT), Parallax is faster than both TF-PS and Horovod.

operations. The slight difference in convergence times of Parallax and Horovod is due to random variable initialization and data shuffling effects unrelated to the techniques described in this paper.

On the other hand, TF-PS is faster than Horovod for the LM model as we expected. For all LM model experiments, Parallax automatically finds a near-optimal number of partitions for sparse variables using its regression-based method. In the case of TF-PS and Horovod, we perform a manual search for the number of partitions as the frameworks do not provide automatic search mechanisms. Thanks to Parallax's hybrid architecture and optimizations such as local aggregation, Parallax achieves a 2.6x speedup compared to TF-PS and a 5.9x speedup compared to Horovod.

Similar to the LM experiments, the NMT model experiments were conducted after applying partitioning of sparse variables for each framework. Parallax converges 2.3x faster than Horovod and 1.7x faster than TF-PS.

6.3 Performance and Scalability

Next, we show the performance of Parallax by comparing the training throughput of Parallax against those of TF-PS and Horovod. Then, we evaluate the scalability of Parallax as we increase the number of GPUs.

Training Throughput Figure 8 shows the training throughput of Parallax, TF-PS and Horovod. According to Figures 8(a) and 8(b), Horovod achieves higher throughput compared to TF-PS on the dense models. For these models, Parallax achieves throughput similar to Horovod. In contrast to the dense models, the three frameworks have significant performance differences for the sparse models. Figures 8(c) and 8(d) depict training throughput for LM and NMT. On 48 GPUs, Parallax shows 2.8x speedup and 2.0x speedup for LM and NMT compared to TF-PS, respectively. Throughout all combinations of the number of machines and different DL models, Parallax always outperforms or gives performance equal to both TF-PS and Horovod.

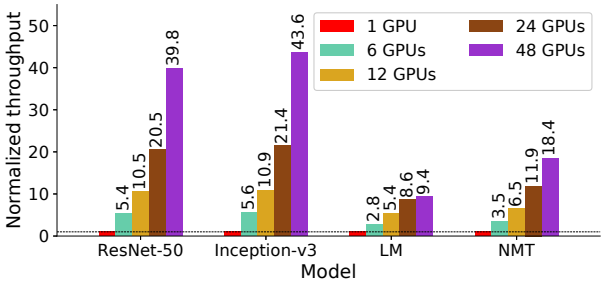


Figure 9. Scalability of Parallax for each model. With 48 GPUs, normalized throughput of Parallax is 39.8, 43.6, 9.4 and 18.4 for ResNet-50, Inception-v3, LM and NMT respectively, which scales better than TF-PS (30.4, 28.6, 3.4, 9.1) and Horovod (39.8, 44.4, 1.6, 6.1)

Models	AR	NaïvePS	OptPS	HYB (AR + OptPS)
LM	45.5k	98.9k	250k	274k
NMT	68.3k	102k	116k	204k

Table 4. Training throughput (words/sec) of various architectures.

Scalability of Parallax. Figure 9 presents the scalability of Parallax for the four models. We define normalized throughput for n GPUs (n = 6, 12, 24, 48) as the ratio of the throughput for n GPUs over the throughput for 1 GPU. Ideally, the normalized throughput should be equal to the number of GPUs. The difference between ideal throughput and actual throughput comes from the added communication overhead for distributed training. For ResNet-50 and Inception-v3, Parallax scales out well, achieving 39.8x and 43.6x speedups on 48 GPUs. The scalability of LM and NMT is worse than ResNet-50 and Inception-v3. The normalized throughput of LM is 9.4, and that of NMT is 18.4 with 48 GPUs. LM and NMT stress more communication than ResNet-50 and Inception-v3 due to the large size of variables and relatively light computation. For example, the number of variable elements exchanged per GPU is 101 million for NMT, 26 million for ResNet-50, and 24 million for Inception-v3.

6.4 Effect of Hybrid Architecture

To analyze the effectiveness of the hybrid architecture compared to employing only one architecture, we compare the throughputs of AR using Parallax, the naïve PS architecture (NaïvePS) using TF-PS, optimized PS (OptPS) in Parallax, and the hybrid architecture (HYB) based on AR and OptPs, as shown in Table 4. OptPS includes local aggregation and smart operation placement across server and worker processes. We experiment for LM and NMT models using 8 machines with 48 GPUs. In the experiments, sparse variable partitioning is applied to all architectures because of the large size of the sparse variables.

As we show in the previous section, NaïvePS (TF-PS) outperforms AR on sparse models - the speedup is 2.2x for LM and 1.5x for NMT. OptPS improves the throughput of NaïvePS by 2.5x and 1.1x for LM and NMT, respectively. The speedup continues on HYB - it is 1.1x faster than OptPS for the LM model and 1.8x faster for the NMT model. HYB's performance improvement is more significant in the NMT model which has a similar ratio of sparse and dense variables (56% of total variables are dense and the remaining 44% are sparse). On the other hand, the speedup of the LM model is relatively low as we progress from OptPS to HYB. The reason is that the majority of variables in the LM model are sparse variables (the size of sparse variables is 99% of the size of total variables), and the effect of optimizing the communication of dense variables by combining AR and PS is rather small.

Models	Parallax	Min	Optimal
LM	274k	96.5k	289.5k
NMT	204k	124.1k	208k

Table 5. Training throughputs (words/sec) from different partitioning methods with 8 machines (48 GPUs). The Parallax column corresponds to Parallax's partitioning method, the Min column shows the results of using the smallest number of partitions possible without memory exceptions, and the Optimal shows the results of the brute-force method.

length	α_{model}	Parallax	TF-PS	Speedup
120	1.0	437k	214k	2.04x
60	0.52	511k	219k	2.33x
30	0.28	536k	221k	2.43x
15	0.16	557k	193k	2.89x
8	0.1	480k	159k	3.02x
4	0.07	285k	94k	3.03x
1	0.04	82k	24k	3.42x

Table 6. The training throughput (words/sec) of Parallax and TF-PS, and speedup of Parallax compared to TF-PS under various sparsity degrees (α_{model}). *length* represents the number of words in a data instance.

6.5 Sparse Variable Partitioning

We present the efficiency of the sparse variable partitioning method of Parallax for LM and NMT in Table 5. The efficiency is measured by comparing throughput of Parallax's method with that of a brute-force method that finds the optimal number of partitions by first starting from the smallest number of partitions possible without memory exceptions (4 and 2 partitions for LM and NMT, respectively) and gradually increasing the number of partitions by 2 to get better throughput. The brute-force method stops searching when the number of partitions is too large that throughput drops more than 10% compared to the highest throughput observed. Compared to the results using the smallest number of partitions without exceeding the memory bound (Min), Parallax's partitioning method improves the throughput by 2.84x and 1.64x for LM and NMT, respectively. Moreover, Parallax's method does not fall behind more than 6% compared to the brute-force method (Optimal). The brute-force method is much more inefficient than Parallax; Parallax spends at most 20 minutes to get sampling results of at most 5 runs while the brute-force method needs to collect results from more than 50 runs.

6.6 Effect of Sparsity Degree

Table 6 compares the training throughput (words/sec) under various sparsity degrees (α_{model}) using Parallax and TF-PS.

All experiments were performed on 48 GPUs using a constructed LM model that uses dense variables and vocabulary smaller than those of the original LM model to test under a wide range of α_{model} values. α_{model} is controlled by the number of words (length) in a data instance with the batch size fixed. The longer the length of a data instance, more elements of sparse variables are utilized at an iteration, thus the larger the value of α_{model}. Parallax has higher throughput than TF-PS for all the sparsity conditions. The fixed cost for dense variable communication is becoming more significant as the amount of data transfer for sparse variables reduces due to the small α_{model}. Therefore, the biggest speedup of Parallax compared to TF-PS is 3.42 when α_{model} is minimum.

7 Related Work

Data Parallel Training on Existing DL Frameworks Existing DL frameworks, such as TensorFlow [1], MXNet [8] and PyTorch [29], support data parallel training with multiple machines and GPUs. However, to the best of our knowledge, none of the existing frameworks consider the sparsity as an important factor of data parallel training, only supporting either the PS architecture or the AR architecture at one time. Moreover, unlike Parallax, most of the existing frameworks make users manually modify single-GPU code to be trainable in a distributed environment.

For example, TensorFlow data parallelization APIs such as `SyncReplicasOptimizer`, `replica_device_setter`, `MonitoredTrainingSession` and `Server` are designed only for the PS architecture. Moreover, these APIs require additional modifications when converting a single-GPU graph to a distributed one, and users are still responsible for debugging if distributed training does not work correctly and efficiently. To handle this issue, TensorFlow introduces a high-level `DistributionStrategy` API as an experimental feature, which removes the manual modification process from users by converting a single-GPU code to a distributed version automatically. However, even with such a high-level API, users must select which strategy to use among various strategies including `MirroredStrategy`, `CollectiveAllReduceStrategy` and `ParameterServerStrategy`, without any clue about the relationship between the model sparsity and training throughput. Additionally, the programming model with `DistributionStrategy` is less flexible than the low-level data parallelization API to achieve automated distribution. The current implementation of `DistributionStrategy`[5] does not support synchronous multi-machine training with the PS architecture, input data sharding API for multi-machine training, and advanced performance optimizations that Parallax provides.

[5]TensorFlow v1.12, November 2018.

MXNet [8] supports data parallel training using a distributed key-value store for data synchronization between machines and GPUs, supporting only the PS architecture without considering the model sparsity. In addition, a single-GPU code should be manually modified to pull variables and to push gradients using the store. Moreover, it is impossible to improve communication efficiency by offloading some computations from a worker to servers with the key-value store. PyTorch [29] supports distributed training only with the AR architecture. PyTorch provides APIs for constructing communication groups, averaging gradients, and adding aggregation methods for data parallel training. Horovod [34] also provides an abstraction of efficient AR algorithms and implementations. Parallax also uses Horovod's MPI operators for TensorFlow including `HorovodAllreduceOp`.

Combining PS Architecture with Other Communication Mechanisms There exist other frameworks that try to improve performance by combining the PS architecture with other communication mechanisms. MXNET-MPI [24] divides GPUs into multiple groups, where GPUs in the same group communicate using `AllReduce/Reduce` operations. Each group then communicates with each other using the PS architecture. For this new architecture, the paper introduces a new MPI Elastic SGD algorithm, which allows synchronous SGD methods within an MPI group and asynchronous SGD methods between groups to mitigate both the network contention problem in synchronous training and the staleness problem in asynchronous training. The mixture of the PS architecture and `AllReduce/Reduce` operations is mainly used for controlling asynchrony for the new algorithm. On the other hand, Parallax combines the PS and AR architectures while maintaining the widely-used algorithm, synchronous SGD. Moreover, since MXNET-MPI still uses collective communication within a group, it requires a larger amount of network transfer for handling sparse variables compared to Parallax.

Poseidon [44] combines the PS architecture and sufficient factor broadcasting (SFB) communication that uses peer-to-peer connections of workers. SFB communicates sufficient factors of a gradient matrix for fully connected (FC) layers using its decomposability as two smaller vectors. Even though Poseidon pursues a similar approach to choose an optimal training architecture based on the estimation of data transfer, it focuses on gradients of the FC layers, while Parallax focuses on sparse variables and their gradients.

Model Parallel Training Model parallelism is another approach to deal with the large, sparse models. In model parallelism, a single model is split across multiple GPUs, and each GPU computes only a part of the model. A problem of model parallel training is underutilization of GPUs due to the small size of each fragment assigned to a GPU. PipeDream [15] addresses the problem using overlapped computation for

multiple iterations. However, the staleness caused by computing multiple iterations in parallel is getting significant if the number of GPUs increases. Recently, hybrid strategies of model-parallelism and data parallelism [41, 42] are introduced to find optimal parallelization methods by considering both sides, but they still need an efficient data parallel training to improve overall performance.

Increasing Variable Sparsity through Network Sparsification A dense model can be converted into a sparse model by employing pruning techniques [22, 23] that are used to reduce the amount of computation, communication, and memory usage for both training and inference. These techniques utilize different subsets of model variables for different inputs, making the variables sparse. Quantization techniques [3, 14, 46] change gradient tensors of dense variables into sparse formats by increasing the number of zero elements in the gradients. Even when the model is intrinsically dense, by applying network pruning or quantization, we believe that Parallax's hybrid architecture can outperform other frameworks that only utilize the PS or AR architecture. We consider exploring this direction as future work.

8 Conclusion

We present Parallax, a framework that provides sparsity-aware data parallel training. Parallax introduces a hybrid approach that combines different training architectures according to the sparsity of variables to reduce the amount of network transfer. Parallax also proposes a method for partitioning sparse variables to maximize parallelism while maintaining low computation and communication overhead. Its automatic graph transformation allows users to use their single-GPU program for training on a distributed environment while maintaining scalable performance. We show that Parallax achieves higher performance and scalability for sparse models compared to TensorFlow and Horovod in a cluster of 48 GPUs. We open sourced Parallax in the hope of facilitating users to take advantage of sparsity-aware data parallel training. Parallax is publicly available at https://github.com/snuspl/parallax.

Acknowledgments

We thank our shepherd Madan Musuvathi and the anonymous reviewers for their insightful comments. This work was supported by the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics), the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test), and Samsung Advanced Institute of Technology.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 265–283.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. (2017). arXiv:1711.04325 https://arxiv.org/abs/1711.04325

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Proceedings of Advances in Neural Information Processing Systems*. Curran Associates, Inc., 1709–1720.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proceedings of the 9th Python in Science Conf*, Vol. 1. 1–7.

[5] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 4960–4964.

[6] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. (2013). arXiv:1312.3005 https://arxiv.org/abs/1312.3005

[7] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting Distributed Synchronous SGD. (2016). arXiv:1604.00981 https://arxiv.org/abs/1604.00981

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. (2015). arXiv:1512.01274 https://arxiv.org/abs/1512.01274

[9] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 571–582.

[10] Jan Chorowski, Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. End-to-end continuous speech recognition using attention-based recurrent NN: first results. (2014). arXiv:1412.1602 http://arxiv.org/abs/1412.1602

[11] Facebook. 2017. Caffe2. https://caffe2.ai

[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. (2017). arXiv:1706.02677 https://arxiv.org/abs/1706.02677

[13] Suyog Gupta, Wei Zhang, and Fei Wang. 2017. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 4854–4858.

[14] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of International Conference on Learning Representations*.

[15] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 770–778.

[17] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. In *Proceedings of Workshop on Machine Learning Systems in The 32th Annual Conference on Neural Information Processing Systems*. IEEE.

[18] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the Limits of Language Modeling. (2016). arXiv:1602.02410v2 https://arxiv.org/abs/1602.02410

[19] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02927

[20] Sameer Kumar, Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, and Ashish Verma. 2017. Efficient Training of Convolutional Neural Nets on Large Distributed Systems. (2017). arXiv:1711.00705 http://arxiv.org/abs/1711.00705

[21] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 583–598.

[22] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime neural pruning. In *Proceedings of Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2181–2191.

[23] Jian-Hao Luo and Jianxin Wu. 2018. AutoPruner: An End-to-End Trainable Filter Pruning Method for Efficient Deep Model Inference. (2018). arXiv:1805.08941 http://arxiv.org/abs/1805.08941

[24] Amith R Mamidala, Georgios Kollias, Chris Ward, and Fausto Artico. 2018. MXNET-MPI: Embedding MPI parallelism in Parameter Server Task Model for scaling Deep Learning. (2018). arXiv:1801.03855 https://arxiv.org/abs/1801.03855

[25] Amith R Mamidala, Jiuxing Liu, and Dhabaleswar K Panda. 2004. Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms. In *Proceedings of International Conference on Cluster Computing*. IEEE, 135–144.

[26] NVIDIA. 2013. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect

[27] NVIDIA. 2017. NCCL. https://developer.nvidia.com/nccl

[28] Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016. Conditional Image Generation with PixelCNN Decoders. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 4797–4805.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[30] Pitch Patarasuk and Xin Yuan. 2007. Bandwidth efficient all-reduce operation on tree topologies. In *Proceedings of 21th International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[31] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3, 211–252.

[33] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[34] Alexander Sergeev and Mike Del Balso. 2018. Horovod. (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[35] Shaohuai Shi and Xiaowen Chu. 2018. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. In *Proceedings of IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing*. IEEE, 949–957.

[36] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2818–2826.

[38] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Workshop on Machine Learning Systems in The 29th Annual Conference on Neural Information Processing Systems*.

[39] Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, and William Gropp. 2010. A simple, pipelined algorithm for large, irregular all-gather problems. *The International Journal of High Performance Computing Applications 24*, 58–68.

[40] Statistical Machine Translation. 2014. wmt. http://www.statmt.org/wmt14

[41] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2018. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. (2018). arXiv:1807.08887 http://arxiv.org/abs/1807.08887

[42] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2018. Unifying Data, Model and Hybrid Parallelism in Deep Learning via Tensor Tiling. (2018). arXiv:1805.04170 http://arxiv.org/abs/1805.04170

[43] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. (2016). arXiv:1609.08144 https://arxiv.org/abs/1609.08144

[44] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 181–193.

[45] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-aware async-SGD for Distributed Deep Learning. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. AAAI Press, 2350–2356.

[46] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. (2017). arXiv:1702.03044 http://arxiv.org/abs/1702.03044