ScaleDNN: Data Movement Aware DNN Training on Multi-GPU

Weizheng Xu
University of Pittsburgh
wex43@pitt.edu

Ashutosh Pattnaik
Penn State University
ashutosh13@gmail.com

Geng Yuan
Northeastern University
yuan.geng@northeastern.edu

Yanzhi Wang
Northeastern University
yanz.wang@northeastern.edu

Youtao Zhang
University of Pittsburgh
zhangyt@cs.pitt.edu

Xulong Tang
University of Pittsburgh
xulongtang@pitt.edu

Abstract—**Training Deep Neural Networks (DNNs) models is a time-consuming process that requires immense amount of data and computation. To this end, GPUs are widely adopted to accelerate the training process. However, the delivered training performance rarely scales with the increase in the number of GPUs. The major reason behind this is the large amount of data movement that prevents the system from providing the GPUs with the required data in a timely fashion. In this paper, we propose ScaleDNN, a framework that systematically and comprehensively investigates and optimizes data-parallel training on two types of multi-GPU systems (PCIe-based and NVLink-based). Specifically, ScaleDNN performs: i) CPU-centric input batch splitting, ii) mini-batch data pre-loading, and iii) model parameter compression to effectively a) reduce the data movement between the CPU and multiple GPUs, and b) hide the data movement overheads by overlapping the data transfer with the GPU computation. Our experimental results show that ScaleDNN achieves up to 39.38%, with an average of 17.96% execution time saving over modern data parallelism on PCIe-based multi-GPU system. The corresponding execution time reduction on NVLink-based multi-GPU system is up to 19.20% with an average of 10.26%.**

Index Terms—**Data Movement, Data Parallelism, Multi-GPU**

I. INTRODUCTION

With the capability to provide unprecedented accuracy levels, Deep Neural Networks (DNNs) have become a popular technique in application domains such as object classification, autonomous vehicles and natural language processing [16], [40], [44]. However, training DNN models is time-consuming. For instance, with 16 Google TPUs, training the BERT network with 512 batch size takes 81.4 hours [6]. Similarly, training DenseNet-40 using a single GPU takes two days to achieve convergence [12].

To accelerate the training process, multi-GPU systems are adopted with DNN data parallelism and model parallelism to provide speedups [19], [21], [24], [33]. However, naively employing data parallelism across multiple GPUs can lead to sub-optimal execution even performance degradation. This is because the data movement overheads significantly increase and may dominate the execution time when multiple GPUs are employed. Specifically, modern DNN data parallelism employs a master GPU to split and distribute mini-batches.
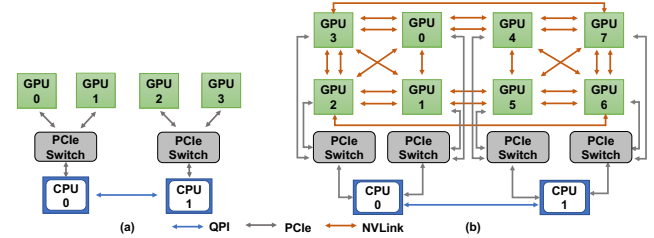

Fig. 1. Interconnections in multi-GPU system. (a) PCIe interconnection and (b) Hybrid Cube-Mesh topology NVLink interconnection

While doing so reduces the CPU load and avoids potential CPU bottlenecks that are caused by frequent API calls (e.g., `cudaMemcpy()`), it introduces extra data movements. As revealed by our characterization (Section III), such data movement is significant when the batch size is large. Furthermore, mini-batch and model parameter (i.e., weight) transfers are on the critical execution path. As a result, GPUs cannot start forward propagation until the required data is in place, leading to the under-utilization of GPU computing resources.

To reduce the data movement overheads, we propose ScaleDNN, a framework that systematically and comprehensively optimizes the data movement to improve the scalability of DNN training on two types of multi-GPU systems (i.e., PCIe-based and NVLink-based). Specifically, ScaleDNN employs CPU-centric input batch distribution and pre-loading to effectively hide the mini-batch transfers by overlapping it with the GPU computation. As such, ScaleDNN removes mini-batch transfer from the execution's critical path and improves GPU utilization. It also implements a dynamic compression mechanism for weights[1] built upon the weight similarity with negligible accuracy impact. This paper contributes as below:

- We conduct an in-depth characterization of various DNN models running data parallelism training in multi-GPU systems and quantitatively demonstrate the overheads caused by excessive data movements. Our characterization is conducted on PCIe-based and NVLink-based multi-GPU systems to reflect the data movement overheads in state-of-the-art multi-GPU systems.

---

[1] In this paper, we use the term "weight" and "model parameter" interchangeably.

- We propose ScaleDNN, a framework that equips three optimizations: *CPU-centric input batch splitting, mini-batch preloading, and model parameter compression*. The first two optimizations employ CPU to dispatch the mini-batches and remove mini-batch transfer from the execution's critical path. The third optimization explores the similarity among model parameters and leverages compression to reduce the data transfer with negligible impact on model accuracy.
- We evaluate ScaleDNN on both PCIe-based and NVLink-based multi-GPU systems. Our experimental results show that ScaleDNN achieves up to 39.38%, with an average of 17.96% execution time saving over modern data parallelism on the PCIe-based multi-GPU system. On the NVLink-based multi-GPU system, it achieves up to 19.20%, with an average of 10.26% execution time saving.

II. BACKGROUND

A. Multi-GPU system architecture

GPUs are widely used in training DNN models [8], [14], [15], [39], [49]. In this paper, we target DNN training on a single-node multi-GPU system as shown in Fig 1. Fig 1(a) shows that, in a two-socket multi-GPU server, each pair of GPUs is connected to a PCI host bridge and the two pairs are separately connected to two CPU sockets. The two CPU sockets are linked with QPI [27]. In such a multi-GPU system, PCIe buses are used to connect CPU and GPUs. Another modern multi-GPU system employs NVLink instead of PCIe to connect GPUs as shown in Fig 1(b). In such a system, different topologies can be used to connect the GPUs such as ring [18], [46], hybrid cube-mesh [1], [22], [41], etc. The example in Fig 1(b) uses the hybrid cube-mesh topology to connect the multiple GPUs, ensuring that a GPU is no more than two hops away from any of the other GPUs.

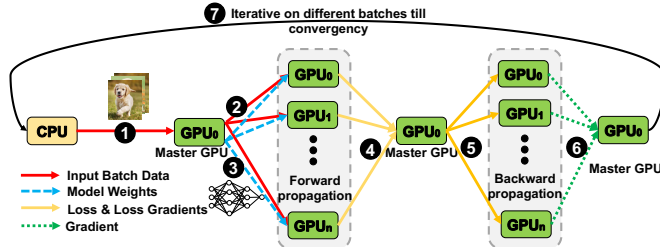### B. Baseline Data Parallelism on Multi-GPU

Fig. 2. Data parallelism training on multi-GPU systems.

The DNN training process consists of multiple epochs and each epoch refers to one pass through the entire input training dataset. In general, the dataset is divided into a number of batches. Processing each batch is called an iteration. Figure 2 shows the dataflow of one iteration. We use the data parallelism in PyTorch [34] as our baseline. As shown in the figure, the batch of input feature maps is first transferred from CPU to the master GPU[2] (❶). Note that, for the weights, only the initial values are generated by the

[2]In data parallelism, the master GPU is responsible for receiving data from CPU, scattering/gathering data to/from peer GPU workers, and updating the model parameters.

CPU and transferred to the master GPU before the first batch starts. All the subsequent weight updates are performed on the master GPU without CPU involvement. The master GPU then splits the input batch into mini-batches and distributes the mini-batches among the peer GPU workers (❷). The master GPU also broadcasts the updated weights to peer GPUs (❸). Upon receiving the mini-batches and updated weights, all the GPUs start forward propagation (FP) and the outputs of FP are gathered on the master GPU to compute the loss (❹). Then, these loss values are sent back to the GPUs (❺) for backward propagation (BP) to compute the updated gradients. Finally, the gradients are gathered back on the master GPU (❻) to update the weights for the next batch processing. The training iterates this process using different batches until convergence is reached (❼). Note that, the data parallelism training involves different types of data movements between the CPU and multiple GPUs as highlighted by the colored lines in the figure. As our focus in this paper is a single node server with multiple GPUs, we adopt the master-GPU based data parallelism training in the PyTorch framework [4], [26]. However, our approach is applicable to distributed multi-node systems and is complementary to distributed data parallelism to provide additional benefits when combined.

III. CHARACTERIZATION AND MOTIVATION

A. Methodology

TABLE I
CHARACTERISTICS OF SERVERS USED IN EXPERIMENTS.

Server name	GPUs per Server	Interconnects	GPU memory
Server-A	4x GTX1080Ti	PCIe	11GB
Server-B	8x V100	PCIe and NVLink	32GB

We use two multi-GPU servers, one with PCIe-based, and another with NVLink-based (summarized in Table I) communication to conduct our characterization as well as to evaluate our proposed optimizations later in Section V. Server-A is equipped with the Intel(R) Xeon(R) Silver 4112 processor. Each processor has two CPU sockets connected with Intel QPI and each CPU socket consists of 4 cores working at 2.60GHz. The node is featured with 96GB RAM. Each node has 4 NVIDIA GTX1080 Ti GPUs (with 11GB on-board GDDR memory) connected via PCIe 3.0.

TABLE II
LIST OF DNN MODELS.

Models	Model Size
VGG19	549MB
VGG16	528MB
AlexNet	233MB
ResNet34	84MB
ResNet50	98MB
MobileNet v2	14MB
MNASNet1_3	24MB
Bert-large	1.3GB

Server-B is equipped with 2 socket Intel Xeon "Cascade Lake" CPUs. Each CPU socket consists of 20 cores working at 2.50-3.90GHZ. The server has 8 V100 GPUs connected using NVLink, each with 32GB of GPU memory. PCIe bus is used to connect the CPU and GPUs. The server is featured with 512GB RAM. All DNNs are trained using modern data parallelism on multiple GPUs that is supported in PyTorch 1.4.1. We use CUDA 10.1 and cuDNN v7.6.5 and NCCL 2.4 [30] libraries for communications (e.g., broadcast and all-reduce operations) across the multiple GPUs. Table II summarizes the eight representative DNN models with diverse
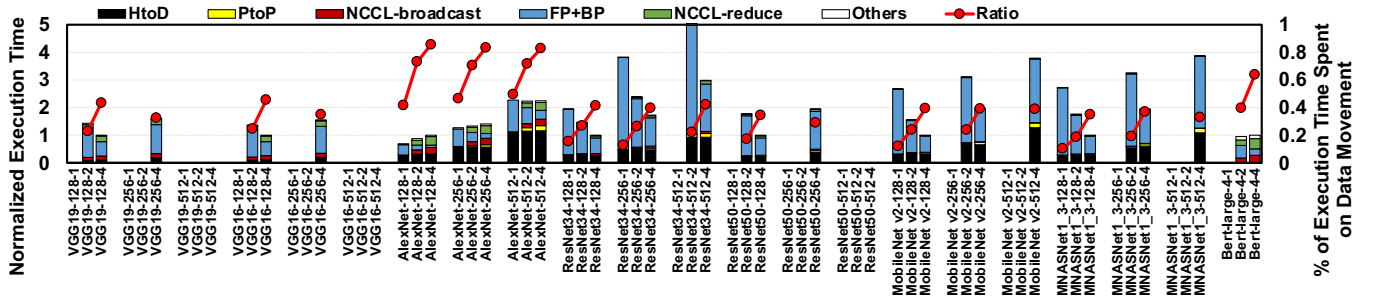
Fig. 3. Normalized execution time of different configurations–**Server A**. The x-axis represents different configurations in the form of `<DNN model, Batch size, Number of GPUs>`. Except Bert, all results are normalized to the `<DNN model, 128, 4>`. Bert is normalized to the `<Bert-large, 4, 4>`. There are two y-axes in the figure. The stacked bars are associated with the left y-axis to represent the normalized execution time. The red lines are associated with the right y-axis to represent the portions of data movement overheads in the overall execution time. The configurations without corresponding stacked bars suffer out-of-memory issues.
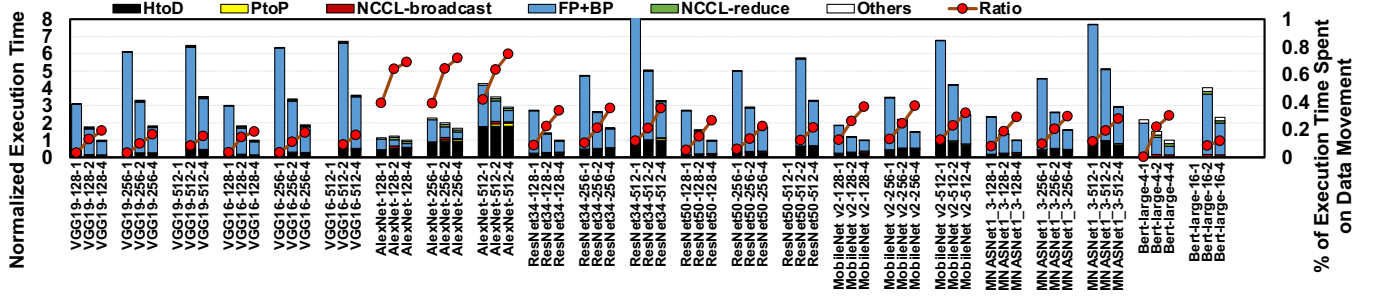

Fig. 4. Normalized execution time of different configurations–**Server B**.

model sizes as benchmarks in our experiments. For image classification tasks (the first seven models in table II), we use Tiny-Imagenet dataset [20] as the training input. The models are trained using the stochastic gradient descent (SGD) algorithm. The dataset applied for the Bert (one representative language model) is SQuAD2.0 [36] and the model is pre-trained. We use three different batch sizes (128, 256, and 512 images) for image classification tasks. For Bert, we use different batch sizes on different servers due to the different GPU memory capacities of GTX 1080Ti and V100. To conduct the characterization, we use the NVIDIA profiler tool (nvprof [31] and Nsight system [32]).

B. Execution Time Breakdown Analysis

In this section, we conduct the detailed execution time breakdown analysis of DNN training on two servers with different interconnections. The results are shown in Figure 3 and Figure 4. The stacked bars represent the execution time breakdown and are associated with the left y-axis. The line graph is associated with the right y-axis and shows the percentage of execution time spent on data movement. Note that, the executions of configurations without stacked bars incur out-of-memory (OOM) errors on GPUs. Due to larger GPU memory in Server B than Server A, Server B has less OOM occurrence. We breakdown the execution time into six categories: 1) transferring input batch of feature maps from CPU to the master GPU (denoted as HtoD[3] in the figure), 2) distributing mini-batches from the master GPU to other GPUs (denoted as PtoP), 3) broadcasting the model parameters

from the master GPU to other GPUs (denoted as NCCL-broadcast), 4) forward propagation and backward propagation (denoted as FP+BP), 5) gathering the gradients after BP to the master GPU (denoted as NCCL-reduce), and 6) others that include initialization, updating weights, etc. Among these six categories, except for the FP+BP and others, the rest four categories involve data movement. From the figures, one can make the following observations:

Observation 1. The total execution time does not proportionally scale with the increasing number of GPUs for all the DNN models and batch sizes (shown in stacked bars). For example, on server A, `ResNet34` with batch size 256 (`<ResNet34, 256, 4>`) only obtains 2.2X execution time reduction on four GPUs compared to running on one GPU (`<ResNet34, 256, 1>`). Other models show a similar trend. In particular, `AlexNet` and `Bert-large` do not benefit from more GPUs and its execution time increases when four GPUs are used. The major reason behind this is the dominating data movement overheads. As shown in the figures, while the data parallelism effectively parallelizes the computation (i.e., FP+BP) across multiple GPUs, the data movement overheads occupy a large portion of the execution time as depicted by the line graph. The average data movement overheads occupy 26.83%, 34.31%, 46.28% of the training time with 1 GPU, 2 GPUs and 4 GPUs, respectively. We also observe from the line graph that the percentage of time spent on transferring data increases when more GPUs are employed for data parallelism. This is because employing more GPUs introduces additional data transfers among CPU and GPUs (e.g., weight broadcasting and mini-batch dispatching), and consequently, spends more time on data movement. Server B results in Figure 4 show similar trends as Server A. The

---

[3]While most HtoD is caused by batch transfer, it also includes a small portion of transferring other marginal data such as input targets, etc.

average data movement overheads on server B occupy 12.75%, 22.81%, 31.64% of the training time with 1 GPU, 2 GPUs and 4 GPUs, respectively. We can observe that server B has less data movement overhead compared with server A due to the adoption of faster interconnection–NVLink. However, the data movement overheads still occupy a sizeable portion of the execution time.

Observation 2. HtoD data transfer occupies the most significant portion of execution time among the aforementioned four types of data movements. Take server A results as an example, on average, HtoD takes 20.28% and 24.51% for 2 GPUs and 4 GPUs, respectively. Further, we can observe that when the input batch size changes (e.g., from 256 to 512), the HtoD transfer time also increases due to the increased data movements involved in transferring mini-batches.

Observations 3. As the number of GPUs increases, additional PtoP data transfer occurs. It can be observed that with larger batch sizes, the PtoP portion is more significant due to the increased size of mini-batches. Intuitively, the PtoP time is expected to be at a similar magnitude of HtoD. However, as shown in the figure, PtoP occupies a smaller portion compared to HtoD. This is because the PtoP transfer has a higher throughput (optimized by NVIDIA APIs) compared to HtoD transfer. Specially, we measure that the throughput of PtoP is around 7-10 times over HtoD transfer throughput.

Observations 4. For models with a large number of weights, NCCL-broadcast and NCCL-reduce overheads also contribute a sizable portion of the training time. In particular, for those 4 large models we studied on server A (i.e., VGG19, VGG16, AlexNet and Bert-large), NCCL-broadcast and NCCL-reduce occupies an average of 24.85%, 35.88% of the training time on 2 GPUs and 4 GPUs, respectively. In fact, as modern DNNs become wider and deeper, the model size increases and the broadcast and the reduce overheads are identified as one of the major bottlenecks that limit the scalability [25].

C. GPU SM Efficiency

To further understand the impact of data movements, we conduct a timeline analysis of GPU utilization on server A. We use sm_efficiency[4] available in the nvprof toolset to represent the GPU utilization. Figure 5 shows the sm_efficiency of four GPUs during a snippet of execution. The figure plots three batches (i.e., epoch 30, batches 100 to 102) from <VGG19, 128, 4> and <ResNet50, 256, 4>. The observed pattern applies to other batches and models as well. The shaded blocks in the figure capture the phases where GPUs are waiting for input batch data and model parameters to be propagated before FP computation and after BP computation. For instance, in <ResNet50, 256, 4>, training three batches takes 1.56s in total where 0.45s (29.05%) spend on waiting for data and all four GPUs are under-utilized. The timeline results reported in Figure 5 is consistent with the percentage reported in the Figure 3, indicating that the data movement is a primary
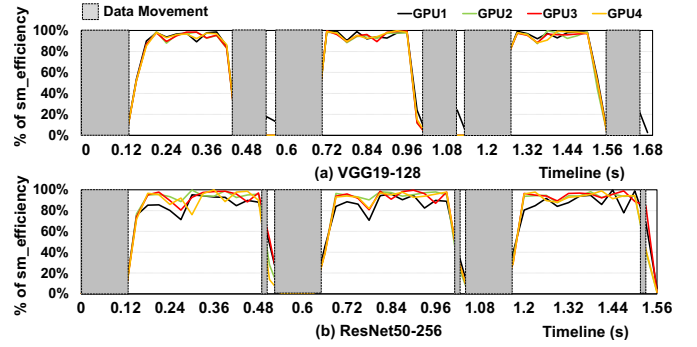
Fig. 5. Timeline results of GPU utilization for three batches during baseline training on Server A. (a) Results for <VGG19, 128, 4>. (b) Results for <ResNet50, 256, 4>.

impediment that prevents the scaling of training DNN models on multi-GPU systems.

D. Takeaway.

Based on the results, it is imperative to optimize/reduce the data movement overheads in order to deliver scalable DNN training performance. Therefore, the implications are twofold. First, it is important to reduce the data movement overheads caused by input batch data as it causes the most significant performance degradation. Second, though the overheads caused by transferring model parameters are relatively small (due to the using of peer to peer GPU transfer), it still occupies sizable portions in large networks (e.g., VGG19, VGG16, AlexNet, Bert-large).

IV. OUR APPROACH: SCALEDNN

In this paper, we aim to i) reduce the data movement overheads caused by transferring input batches and model parameters (i.e., weights), ii) remove data transfers from the execution's critical path, and iii) improve the GPU utilization for data parallelism training. To this end, we propose ScaleDNN which consists of three optimizations to tackle the challenges and achieve our goals. Specifically, ScaleDNN implements i) CPU-centric input batch splitting, ii) mini-batch pre-loading, and iii) model parameter compression. In the rest of this section, we discuss each optimization in detail.

A. CPU-centric Batch Splitting

Fig. 6. CPU-centric batch splitting.

Figure 6(a) depicts the baseline input batch dispatching. Such a data transfer flow has two disadvantages. First, the volume of data transferred increases with more GPUs. For instance, let us assume the batch size is n MBs, the number of GPU is x, and the mini-batch size is n/x MBs. The whole batch is first transferred to the master GPU and split into mini-batches which are then distributed to $x - 1$ peer GPUs. Totally, the baseline mechanism transfers $n + (x - 1) \times n/x$ MBs. Therefore, with more GPUs (x) and fixed batch size

---

[4]we use sm_efficiency which is defined as the percentage of time that the streaming multiprocessor (SM) has at least one warp that is active.
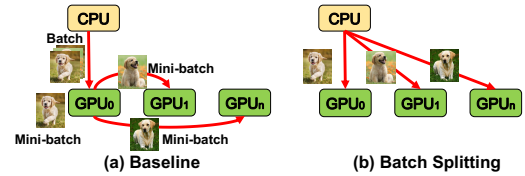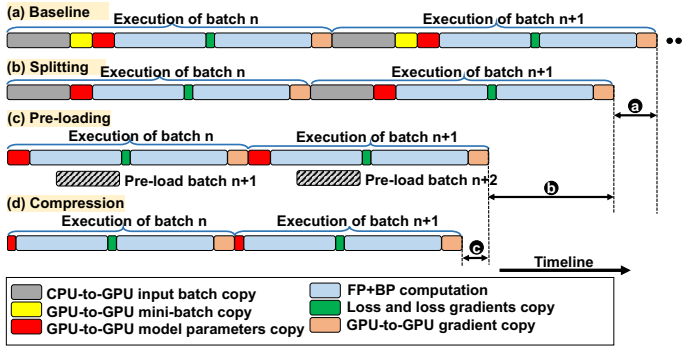
Fig. 7. The execution time reduction brought by our proposed optimizations compared to the baseline execution.

(n), $(x-1)/x$ is larger and finally leads to more data transfer. Second, the master GPU may potentially become the bottleneck. Additionally, since modern GPUs have limited memory capacity compared to CPUs, employing a master GPU is more likely to hit the bounds of GPU memory capacity as the master GPU will receive the entire batch, leading to more memory consumption compared to other peer GPU workers.

Figure 6(b) shows the proposed CPU-centric mini-batch dispatch mechanism. The number of mini-batches is determined by the number of GPUs in the system that is performing FP and BP computation. By employing CPU-centric mini-batch dispatching, there is no need to have a master GPU for mini-batch dispatching[5]. As a result, the time spent on transferring mini-batches among multiple GPUs (referred to as PtoP data transfer in Figure 3) can be completely removed. Meanwhile, the amount of data movement is fixed when the batch size is fixed, and does not increase with the number of GPUs. We also remove the need to hold the entire batch data in the master GPU memory. We use Figure 7 to illustrate the execution time savings brought by ScaleDNN. As shown in Figure 7(b), the "GPU-to-GPU mini-batch copy" is completely removed from the execution time compared to the baseline in Figure 7(a). As a result, the proposed CPU-centric mini-batch dispatching provides ⓐ saving in the execution time.

B. Mini-batch Pre-loading

Another benefit brought by CPU-centric input batch splitting is to enable the pre-loading of subsequent batches. Recall the observations from Section III, the HtoD batch movement occupies a large portion of training time. The fundamental reason behind this is that the GPUs need to wait for the mini-batches to arrive before starting the FP computation. To reduce the waiting time, ScaleDNN implements mini-batch pre-loading as depicted in Figure 7(c). The key idea is to "pre-load" the mini-batches of batch $n+1$ to overlap it with the computation (i.e., FP+BP) of batch n. As a result, by the time when the GPUs start processing batch $n+1$, the mini-batches have already been pre-loaded to the GPU memory, eliminating the need to wait for the mini-batch transfer. The pre-loading is implemented by launching new CUDA streams for each GPU device to transfer mini-batch

[5]The master GPU still handles the update of model parameters.

data[6]. We also employ asynchronous memory copy APIs to enable the potential overlapping between data transfer and computation. Figure 7(c) shows the execution time saving with mini-batch pre-loading in addition to the mini-batch split optimization. As one can observe from Figure 7(c), with the proposed mini-batch pre-loading, ScaleDNN is able to save an additional ⓑ time. While such pre-loading requires the GPU to pre-allocate and reserve certain memory space for the mini-batches of the next batch, the overheads are marginal. For example, with 224x224 input image resolution, one mini-batch (of 512 batch size) only occupies 73.5MB memory when parallelized among 4 GPUs. Our evaluation results reported later in Section V include these overheads.

C. Model Parameter Compression

While the previous two optimizations focus on reducing the data movement overheads caused by transferring input batch data, it is also important to reduce the overhead caused by transferring model weights, since it also occupies a sizable portion (35.88% on average with 4 GPUs) of execution time while training large DNN models (e.g., VGG19, VGG16, AlexNet and Bert-large in Figure 3).

Compressibility of model weights. To reduce the size of model weights, we investigate the potential of using compression on it. The model weights are stored in the IEEE-754 single-precision floating-point (FP32) format in the baseline execution (as shown in Figure 8(a)). In an IEEE-754 representation, the 32 bits are split into 1 sign bit, 8 exponent bits, and 23 mantissa bits. In this representation, the value's significant bits are captured by the higher bits (i.e., the sign bit, the exponent bit, and the higher bits in the mantissa). We conduct a study to show the similarity of exponents bits among the weights, and the value discrepancy of removing the tailing bits of mantissa. In Figure 9(a), the x-axis represents the possible values of the 8 exponent bits from VGG19 model parameters. The y-axis represents the CDF distribution of the exponent values. We use the weights averaged from all iterations. We observe that most of the exponent values are clustered in a small range of values (from 112 to 124). The values in this range (112-124) share the same higher order 4 bits (0111XXXX). This observation reveals that the potential to use base-delta compression to store the higher 4 bits as base and the lower 4 bits as delta. The observed pattern applies to all other DNNs as well. Figure 9(b) shows the PDF distribution of the value discrepancy between the original value and the value after removing the lower 18 bits from mantissa. The data is collected from model parameters of VGG19. We use the same batches as Figure 9(a) to show value discrepancy. One can further observe from Figure 9(b) that removing the lower 18 bits has a negligible impact on the majority of the values (i.e., ~99% of the values are clustered in a small range of value from -0.0005 to 0.0005). This is because the significant bits

[6]A CUDA stream is a sequence of device operations that execute on the GPU device in order. Operations from different streams may execute concurrently depends on hardware resource availability.
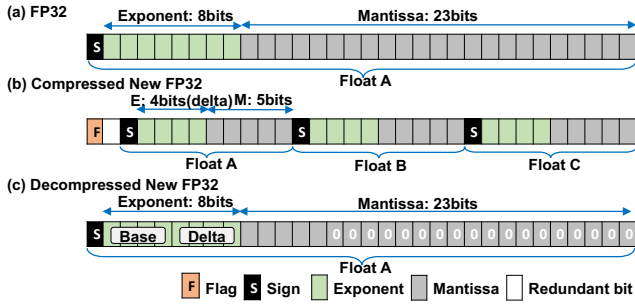
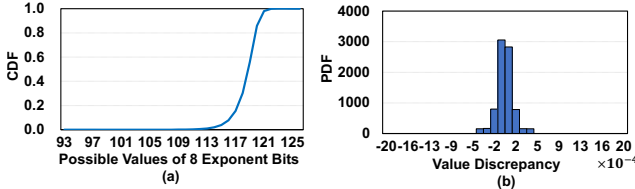Fig. 8. Compression and decompression formats of weights.

Fig. 9. (a) Cumulative Distribution Function (CDF) of exponent bits in weights. (b) Probability Density Function (PDF) of value discrepancy after removing the lower 18 bits in mantissa.

are carried in higher bits. Due to space constraints we only show two DNN models to motivate the compressibility of the weight data, however, we observed similar behaviour in other DNN models as well. The intuition of these observations is similar to weight quantization. Furthermore, we quantitatively compare our approach with quantization in Section V-D.

Based on these observations, we develop a simple-yet-effective compression method to significantly reduce the volume of model parameters being transferred among GPUs. Figure 8(b) shows the format of compressing three FP32 values into one FP32. For the 8 exponent bits, we split them into two parts: higher 4 bits as the base and lower 4 bits as the delta. We use two bases (0111 and 0110) for exponent bits since they are able to capture 99% of the exponent values. We set a flag bit (shown in Figure 8(b)) to denote which base is used in the current compression of the three floating-point numbers. If all three compressed values share the same exponent bits 0111, then we set the flag bit to 1. Otherwise, the flag bit is set to 0 indicating the base 0110 is selected. For the mantissa bits, we remove the lower 18 bits from the value. The new format of the compressed data includes 1 bit for base selection and 10 bits for each compressed floating-point number. As a result, our compression ratio is 3. Figure 8(c) shows the decompression results where the received FP32 is split to restore the three values. Depending on the flag bit, a different base is selected and concatenated with the delta to form the original exponent bits. Our compression is lossy since we set 0s for the lower 18 bits when decompression.

At the beginning of each batch processing, the model weights after synchronization are compressed by the master GPU before broadcasting the model to peer GPUs. Upon receiving, each GPU worker decompresses the parameters and start their forward propagation. With our compression, the data movement overheads of model parameters are reduced, bringing additional ❻ time saving as illustrated in Figure 7. Note that, the compression and decompression processes add

runtime overheads to the batch processing. Therefore, it may happen that training small models with fewer parameters will encounter marginal gains or even performance degradation. To minimize the overheads, we leverage GPUs to perform parallel compression and decompression. As later demonstrated by the experimental results, our proposed compression and decompression significantly reduces the model parameter copy time for DNN models with a large number of parameters, while it has negligible overheads for those models with few parameters.

V. EVALUATION

A. Overall Performance

We use the same set of DNN models and system configuration in characterization to evaluate the proposed ScaleDNN framework. We also compare our work with the state-of-the-art data parallelism DNN training framework–Horovod to address our contributions [38]. Horovod takes advantage of NCCL (one collective communication library) to synchronize and average the gradients among multi-GPU. It also proposes to use FP16 as the compression format for gradients during synchronization. Figure 10 and Figure 11 plot the normalized execution time with 4 GPUs setup (i.e., configuration `<Model, Batch Size, 4>`) on two kinds of servers (one is fully equipped with PCIe bus, the other uses NVLink for GPU-GPU communication). Results are normalized to 4 GPUs baseline execution without applying our optimizations. We only show 4 GPUs results as the trends for 2 GPUs and 3 GPUs are similar. For each DNN model, the first bar represents the results with only splitting and pre-loading, the second bar represents the results with all three optimizations in ScaleDNN, the third bar represents the Horovod results.

From the result in Figure 10, first, we observe up to 39.38%, with an average of 17.96%, execution time saving compared to baseline data parallelism training across the eight DNNs we evaluated. Compared to Horovod using FP16 for gradient compression on large models, our approach achieves an average of 13.13% execution time saving by splitting, pre-loading and weight compression. This observation further reflects our effectiveness in removing data movement overheads. Further, we have two stack bars named "average" to show the averaged normalized execution times of large models and small models, respectively. We observe that with splitting and pre-loading, ScaleDNN achieves an average of 20.02% execution time saving on small models, and 10.29% execution time saving on large models. This is because these four small models have a large HtoD time ratio (as discussed earlier in Figure 3), providing us significant benefit from overlapping a large portion of HtoD transfer time with GPU computation. Second, large batch sizes have more benefits comparing to small batch sizes as more data movement overheads have been reduced. An exception is `AlexNet` whose computation occupies a small portion of execution time and having more GPUs increases the execution time as we discussed in Section III. Third, the proposed weight compression effectively reduces the weight transfer time and further improves the performance of large
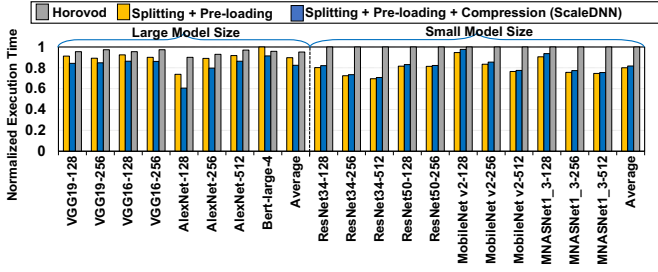
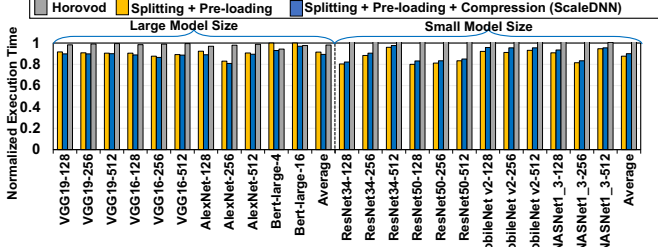Fig. 10. Overall Normalized Execution Time–**Server-A**.

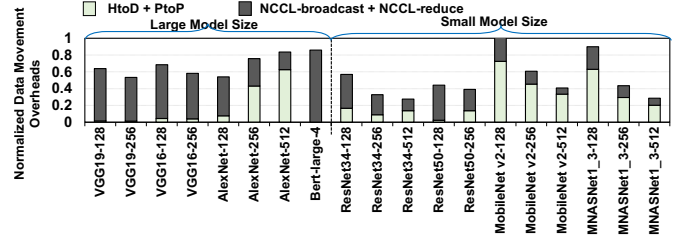Fig. 11. Overall Normalized Execution Time–**Server-B**.

Fig. 12. Normalized data movement overheads–**Server-A**.

Fig. 13. Timeline results of GPU utilization in ScaleDNN on Server A. (a) Results for <`VGG19`, `128`, `4`>. (b) Results for <`ResNet50`, `256`, `4`>.

models. For instance, we observe an average of 7.22% of additional time saving for `VGG19`, `VGG16`, `AlexNet` and `Bert-large` with the overheads included. However, for small model sizes (less than 100MB in Table II), the compression and decompression overheads offset the savings, leading to 1.64% overheads. The results with compression overheads for all models are reported to provide a fair comparison. However, one can easily turn off the compression by setting a command flag in ScaleDNN framework for small models.

The results on server B show an average of 10.26% execution time saving compared to baseline data parallelism training across the eight DNNs we evaluated. Note that, the performance improvement on server B is less than server A. This is because we use high-performance server with NVLink to transfer the data. In addition, as the batch size increases, the improvement trend on server B is different from server A. For instance, on server B, the improvement `ResNet34` value decreases as the batch size increases. This is because, server B has less computation time than server A (due to the more powerful V100 GPU computing capability) but the time spent on transferring input data is the same (both of them use PCIe buses to connect CPU and GPUs), leading to overlapping incompletely. We also conduct the experiments of models `VGG19`, `VGG16`, `ResNet50` and `MobileNet v2` with 8 GPUs and batch size 512. The results show an average of 14.11% execution time saving compared to baseline data parallelism training, indicating that our approach is effective in any number of GPUs.

Figure 12 shows the reduced data movement overheads in ScaleDNN on server A. Similar trend is observed on server B as well. Results are normalized to the data movement overheads in baseline configuration (<`Model`, `Batch Size`, `4`>). As one can observe, ScaleDNN significantly reduces the data movement overheads with an average of 41.55%. Specifically, HtoD+PtoP of `VGG19`, `VGG16`, `ResNet34` and `ResNet50` are almost completely hidden in ScaleDNN. The
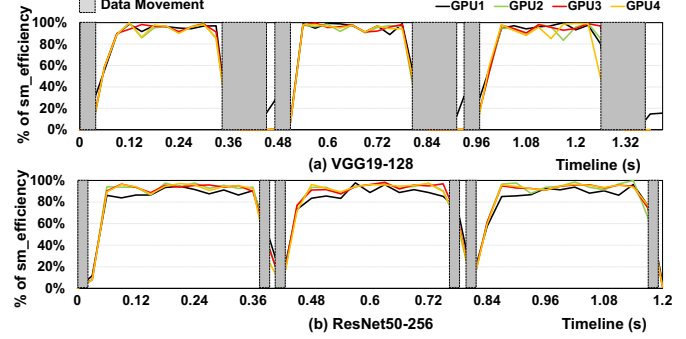
few HtoD+PtoP cannot be overlapped is because of data transfers other than input batch (e.g., input targets used in loss computation) as we mentioned in Section III-B. We further observe that, for `AlexNet`, the normalized HtoD+PtoP increases when the batch size increases. This is because the computation in `AlexNet` is comparably fast and cannot hide the large data movement overheads.

B. GPU SM Efficiency

Figure 13 shows the sm_efficiency on server A to further explain the effectiveness of ScaleDNN. The figure uses the same models from Figure 5. Comparing these two figures, we observe that the data transfer overheads (i.e., denoted using shaded boxes) where sm_efficiency is low have been significantly reduced, indicating that the execution time spent on data movement is saved and overall GPU utilization is improved. As a result, the processing of three batches finishes earlier. Using `VGG19` as an example, the three batch execution consumes 1.38s in Figure 13(a) compared to baseline 1.68s in Figure 5(a). Note that, there still exists a portion of unoptimized data movement overheads. This portion is caused by NCCL-reduce operation where compression is not applicable. We leave the optimization of NCCL-reduce to our future work.

C. Accuracy

Table III shows the top-1 accuracy and top-5 accuracy of baseline training and ScaleDNN under 4 GPU setup on server A. The results on server B are the same. We see that despite using lossy compression, ScaleDNN has a negligible (less than 1%) impact on model accuracy across all DNN models. We also show the over time accuracy of two example DNNs in Figure 14. As one can observe, ScaleDNN has little impact on accuracy over epochs compared to the baseline, indicating the optimizations in ScaleDNN do not compromise the model accuracy. We test the loss value of baseline training

| Model	Top-1 Accuarcy (%)		Top-5 Accuracy (%)		No. of Epochs
	ScaleDNN	Baseline	ScaleDNN	Baseline	
VGG19-128	54.52	54.86	78.02	78.28	80
VGG19-256	46.26	47.07	71.60	72.41	80
VGG16-128	55.73	55.81	78.76	78.84	80
VGG16-256	47.93	48.33	73.21	73.64	80
AlexNet-128	51.78	52.27	76.18	76.40	80
AlexNet-256	47.28	48.10	72.60	73.42	80
AlexNet-512	38.76	38.87	65.08	65.34	80
ResNet34-128	79.66	80.39	91.73	92.09	80
ResNet34-256	79.75	80.51	91.78	92.18	80
ResNet34-512	77.47	78.03	90.04	91.10	80
ResNet50-128	63.47	64.36	83.80	84.55	80
ResNet50-256	58.30	58.73	80.66	81.00	80
MobileNet v2-128	50.40	51.27	74.71	75.64	80
MobileNet v2-256	49.43	50.47	74.16	75.09	80
MobileNet v2-512	41.00	41.61	67.06	67.67	80
MNASNet1_3-128	66.60	67.21	85.73	86.18	80
MNASNet1_3-256	74.61	75.32	89.62	90.10	80
MNASNet1_3-512	74.91	75.79	89.68	90.37	80
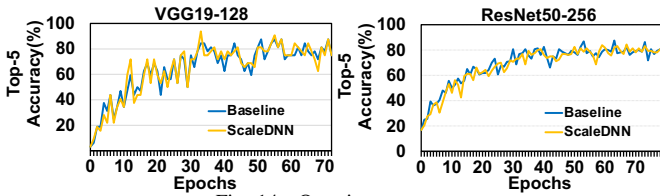
Fig. 14. Overtime accuracy.

and ScaleDNN under 4 GPU setup on `Bert-large` model. We can also observe that optimizations in ScaleDNN do not compromise the model loss.

D. Comparison with Quantization

Recall that the compression in scaleDNN is lossy and the intuition behind is similar to weight quantization. `BFloat16` [17], [47] has been proposed to be a state-of-the-art customized quantization data format for deep learning. It works as well as the FP32 format while delivering increased performance and reducing memory usage. In this work, we implement BFloat16 support on GPUs and quantitatively compare the training performance with ScaleDNN. Figure 15 plots NCCL-broadcast times of three large models (i.e., `VGG19`, `VGG16`, `AlexNet` and `Bert-large`) with 4 GPUs normalized to the baseline 4 GPUs NCCL-broadcast time. We use 256 batch size in comparison because the NCCL-broadcast time does not change with the input batch size. For each model, the

Fig. 15. Comparison between ScaleDNN and BFloat16 on server A.

red bar represents ScaleDNN, and the black bar represents the BFloat16 results. From the figure, we observe that ScaleDNN has better performance than BFloat16 due to the higher compression ratio and less model transfer overheads in ScaleDNN. On average, ScaleDNN achieves an average of 45.77% time saving on NCCL-broadcast, whereas BFloat16 obtains 26.98% time saving. We also want to mention that for small models, both ScaleDNN and BFloat16 does not provide any significant
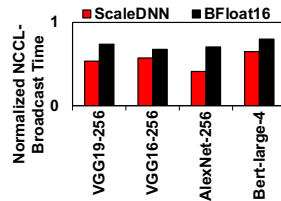
saving in NCCL-broadcast time. Both approaches have a negligible impact on model accuracy (less than 1%).

VI. RELATED WORK

Multi-GPU Parallel Training Previous works have explored data parallelism optimizations to reduce training time [13], [23], [28], [35]. Krizhevsky et al. [19] used data parallelism for convolutional and pooling layers and model parallelism for fully-connected layers. Goyal et al. [7] proposed a three-step all-reduce operation to optimize communication across devices and aggressively overlap gradient synchronization with backward propagation. Pal et al. [33] explored hybrid parallelization to overcome the statistical efficiency losses that data-parallel training incurs at scale. Compared to these prior efforts, we investigate different types of data movements involved in data parallelism on two types of multi-GPU systems (PCIe-based and NVLink-based). Our approach leverages GPU parallelization to hide the data transfer and efficiently compresses the redundancy in weights. Our approach is also orthogonal to prior communication bandwidth optimization [9] and can be combined with these works to further reduce training time.

Data Movements on Multi-GPU Many researchers have contributed to reducing data movement overheads [2], [3], [5], [10], [11], [29], [37], [42], [43], [45], [48] because data movement is one of the key challenges in data parallelism. In these prior works, model compression has been proven as an effective way for DNN models. For example, the parameter pruning [10] and quantization [3] based methods explore the removal of redundant parameters that make a negligible influence on the performance. The knowledge distillation [11] transfers knowledge from a large model to a smaller one. However, they require modifications of the network architectures and cause ineffective training on multi-GPU platforms. Our approach uses simple but effective techniques without any hardware or architecture changes to accelerate the training.

VII. CONCLUSION

DNN training on multi-GPU systems encounters a severe performance burden due to expensive data movement overheads. In this paper, we propose ScaleDNN, a framework that systematically and comprehensively exploits and optimizes the data movement (including both input batches and model weights) to improve the scalability of training on two types of multi-GPU systems (PCIe-based and NVLink-based). Our experimental results show that ScaleDNN achieves up to 39.38%, with an average of 17.96% execution time saving over modern data parallelism on the PCIe-based multi-GPU system. On the NVLink-based multi-GPU system, we can also achieve up to 19.20%, with an average of 10.26% execution time saving over baseline.

REFERENCES

[1] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder, "Topology-aware gpu scheduling for learning workloads in cloud environments," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[2] 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 *AAAI*, 2018.

[3] Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," *arXiv preprint arXiv:1612.01543*, 2016.

[4] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[5] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," *arXiv preprint arXiv:1306.0543*, 2013.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] P. Goyal, P. Dollár, 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.

[8] J. L. Greathouse and G. H. Loh, "Machine learning for performance and power modeling of heterogeneous systems," in *ICCAD*, 2018.

[9] S. Han and W. J. Dally, "Bandwidth-efficient deep learning," in *DAC*, 2018.

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

[11] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[12] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "Densenet: Implementing efficient convnet descriptor pyramids," *arXiv preprint arXiv:1404.1869*, 2014.

[13] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *SysML*, 2019.

[14] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters," in *OSDI*, 2020.

[15] B. K. Joardar, J. R. Doppa, P. P. Pande, H. Li, and K. Chakrabarty, "Accured: High accuracy training of cnns on reram/gpu heterogeneous 3-d architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[16] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado *et al.*, "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Transactions of the Association for Computational Linguistics*, 2017.

[17] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[18] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-gpu system design with memory networks," in *MICRO*, 2014.

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

[20] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge," *CS 231N*, 2015.

[21] J. Lee, I. Hwang, S. Shah, and M. Cho, "Flexreduce: Flexible all-reduce for distributed deep learning on asymmetric network topology," in *DAC*, 2020.

[22] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE TPDS*, 2019.

[23] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: distributed data-parallelism for existing ml applications," in *Eurosys*, 2015.

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[25] Q. Luo, J. He, Y. Zhuo, and X. Qian, "Prague: High-performance heterogeneity-aware asynchronous decentralized training," in *ASPLOS*, 2020.

[26] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, 2020.

[27] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio, "Quickpath interconnect (qpi) design and analysis in high speed servers," in *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*, 2010.

[28] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *SOSP*, 2019.

[29] B. Nicolae, "Datastates: Towards lightweight data models for deep learning," in *Smoky Mountains Computational Sciences and Engineering Conference*, 2020.

[30] NVIDIA, "NVIDIA Collective Communication Library (NCCL) Documentation," 2019.

[31] NVIDIA, "Profiler User's Guide," 2019.

[32] NVIDIA, "Nsight Systems User Guide," 2021.

[33] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta, "Optimizing multi-gpu parallelization strategies for deep learning training," *IEEE Micro*, 2019.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.

[35] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC*, 2020.

[36] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," *arXiv preprint arXiv:1806.03822*, 2018.

[37] B. Reagan, U. Gupta, B. Adolf, M. Mitzenmacher, A. Rush, G.-Y. Wei, and D. Brooks, "Weightless: Lossy weight encoding for deep neural network compression," in *International Conference on Machine Learning*, 2018.

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

[39] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1703.09039

[40] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *NIPS*, 2013.

[41] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-node gpu interconnects for deep learning workloads," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2017.

[42] X. Tang, M. T. Kandemir, H. Zhao, M. Jung, and M. Karakoy, "Computing with near data," in *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2019.

[43] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy, "Data movement aware computation partitioning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[44] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018.

[45] F. Tung and G. Mori, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7873–7882.

[46] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 172–186. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/43ec517d68b6edd3015b3edc9a11367b-Paper.pdf

[47] S. Wang and P. Kanwar, "Bfloat16: the secret to high performance on cloud tpus," *Google Cloud Blog*, 2019.

[48] W. Xu, Y. Zhang, and X. Tang, "Parallelizing dnn training on gpus: Challenges and opportunities," in *Companion Proceedings of the Web Conference 2021*, 2021.

[49] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu," in *AAAI*, 2019.