Hindawi
Applied Computational Intelligence and Soft Computing
Volume 2021, Article ID 3839543, 16 pages
https://doi.org/10.1155/2021/3839543

Research Article

Pipelined Training with Stale Weights in Deep Convolutional Neural Networks

Lifu Zhang and Tarek S. Abdelrahman

Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada

Correspondence should be addressed to Tarek S. Abdelrahman; tsa@eecg.toronto.edu

Received 24 April 2021; Revised 28 August 2021; Accepted 31 August 2021; Published 22 September 2021

Academic Editor: Mehdi Keshavarz-Ghorabaee

The growth in size and complexity of convolutional neural networks (CNNs) is forcing the partitioning of a network across multiple accelerators during training and pipelining of backpropagation computations over these accelerators. Pipelining results in the use of stale weights. Existing approaches to pipelined training avoid or limit the use of stale weights with techniques that either underutilize accelerators or increase training memory footprint. This paper contributes a pipelined backpropagation scheme that uses stale weights to maximize accelerator utilization and keep memory overhead modest. It explores the impact of stale weights on the statistical efficiency and performance using 4 CNNs (LeNet-5, AlexNet, VGG, and ResNet) and shows that when pipelining is introduced in early layers, training with stale weights converges and results in models with comparable inference accuracies to those resulting from nonpipelined training (a drop in accuracy of 0.4%, 4%, 0.83%, and 1.45% for the 4 networks, respectively). However, when pipelining is deeper in the network, inference accuracies drop significantly (up to 12% for VGG and 8.5% for ResNet-20). The paper also contributes a hybrid training scheme that combines pipelined with nonpipelined training to address this drop. The potential for performance improvement of the proposed scheme is demonstrated with a proof-of-concept pipelined backpropagation implementation in PyTorch on 2 GPUs using ResNet-56/110/224/362, achieving speedups of up to 1.8X over a 1-GPU baseline.

1. Introduction

Machine learning (ML), in particular convolutional neural networks (CNNs), has advanced at an exponential rate over the last few years, enabled by the availability of high-performance computing devices and the abundance of data. Today, CNNs are applied in a variety of fields, including computer vision [1], biological and medical science [2], social media [3], image analysis and classification [4, 5], and urban planning [6] to name a few.

However, modern CNNs have grown in size and complexity to demand considerable memory and computational resources, particularly for training. This growth makes it sometimes difficult to train an entire network with a single accelerator [7–9]. Instead, the network is partitioned among multiple accelerators, typically by distributing its layers among the available accelerators, as shown in Figure 1 for an example 8-layer network. The 8 layers are divided into 4 computationally balanced partitions, $P_0, \ldots, P_3$, and each partition is mapped to one of the 4 accelerators, $A_0, \ldots, A_3$. Each accelerator is responsible for the computations associated with the layers mapped to it.

However, the nature of the backpropagation algorithm used to train CNNs [10] is that the computations of a layer are performed only after the computations of the preceding layer in the forward pass of the algorithm and only after the computations of the succeeding layer in the backward pass. Further, the computations for one batch of input data are only performed after the computations of the preceding batch have updated the parameters (i.e., weights) of the network. These dependences underutilize the accelerators, as shown by the space-time diagram in Figure 2; only one accelerator can be active at any given point in time.

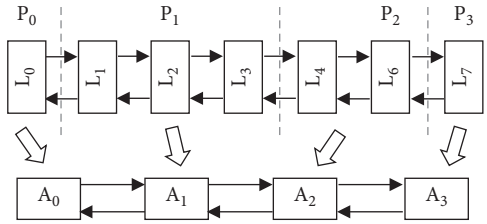

Figure 1: Partitioning of layers.

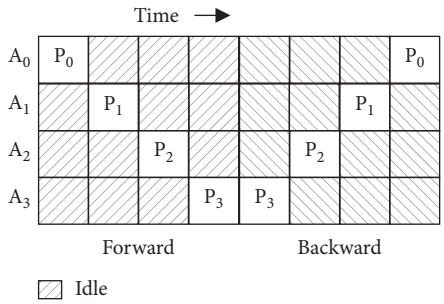

Figure 2: Schedule of computations.

The underutilization of accelerators can be alleviated by pipelining the computations of the backpropagation algorithm over the accelerators [7–9, 11, 12], that is, by overlapping the computations of different input batches on the multiple accelerators. However, this overlap causes an accelerator to potentially use weights that are yet to be updated by an accelerator further down in the pipeline. The use of such stale weights can negatively affect the statistical efficiency of the network, prevent the convergence of training, or produce a model with lower inference accuracy [7–9, 11, 12].

Existing pipelined training approaches either avoid the use of stale weights (e.g., with the use of microbatches [8]), constrain the training to ensure the consistency of the weights within an accelerator (e.g., using weight stashing [9]), utilize weight adjustments (e.g., weight prediction [11]), or limit the use of pipelining to very small networks (e.g., [13]). However, these approaches underutilize accelerators [8], inflate memory usage to stash multiple copies of weights [9], or are unable to handle large networks [13].

In this work, we explore pipelining that allows for the full utilization of accelerators while using stale weights. This results in a pipelining scheme that, compared to existing schemes, is simpler to implement, fully utilizes the accelerators, and has lower memory overhead. We evaluate this pipelining scheme using 4 CNNs: LeNet-5 (trained on MNIST), AlexNet, VGG, and ResNet (all trained on CIFAR-10). These CNNs are commonly used in the literature for the evaluation of pipelined training, and they represent models with a wide range of parameter sizes and complexity. We analyze the impact of weight staleness and show that if pipelining is introduced in early layers in the network, training does converge and the quality of the resulting models is comparable to that of models obtained with nonpipelined training. For the 4 networks, the drop in accuracy is 0.4%, 4%, 0.83%, and 1.45%, respectively.

However, inference accuracies drop significantly when the pipelining is deeper in the network, up to 12% for VGG and 8.5% for ResNet-20. This drop makes the pipelined-trained models inferior to ones trained without pipelining. On the one hand, limiting pipelining to early layers is often not a limitation since the early convolutional layers in the network typically contribute to the bulk of the computations and thus are the ones to use and benefit from pipelining. On the other hand, we also address this drop in accuracy by a hybrid scheme that combines pipelined and nonpipelined training to maintain inference accuracy while still delivering performance improvements.

We demonstrate the potential of our approach to pipelined training using ResNet-56/110/224/362 trained on CIFAR-10 and CIFAR-100 with PyTorch on a 2-GPU system. We show that our pipelined training delivers a speedup of up to 1.8X with only a drop of no more than about 2-3% in inference accuracy.

Thus, this work makes the following contributions:

(1) It proposes and evaluates a pipelined training scheme that uses stale weights. It studies when the use of such stale weights can result in models with comparable prediction accuracies to those produced by nonpipelined training. The result is a simpler and less memory-intensive pipelined training scheme.

(2) It proposes and evaluates a hybrid pipelined training scheme that combines pipelined and nonpipelined training to mitigate loss of prediction accuracies with deeper pipelining.

(3) It presents a theoretical analysis that shows that our pipelined training scheme converges.

The remainder of this paper is organized as follows. Section 2 briefly describes the backpropagation for training of CNNs. Section 3 reviews the current literature on pipelined training. Section 4 details our pipelining scheme and how nonpipelined backpropagation and pipelined backpropagation are combined. Section 5 highlights some of the implementation details. Experimental evaluation is presented in Section 6. Finally, Section 7 gives concluding remarks and directions for future work. A set of appendices provide the training hyperparameters, more detailed results on memory usage, and a proof of convergence for our scheme.

2. The Backpropagation Algorithm

The backpropagation algorithm [10] consists of two passes: a forward pass that calculates the output error and a backward pass that calculates the error gradients and updates the weights of the network. The two passes are performed for input data one minibatch at a time.

In the forward pass, a minibatch is fed into the network, propagating from the first to the last layer. At each layer l, the activations of the layer, denoted by $\mathbf{x}^{(l)}$, are computed using the weights of the layer, denoted by $\mathbf{W}^{(l)}$. When the output of the network (layer L) $\mathbf{x}^{(L)}$ is produced, it is used with the true data label to obtain a training error e for the minibatch.

In the backward pass, the error e is propagated from the last to the first layer. The error gradients with respect to preactivations of layer l, denoted by $\delta^{(l)}$, are calculated. Further, the error gradients with respect to weights of layer l, $\partial e/\partial \mathbf{W}^{(l)}$, are computed using the activations from layer $l-1$ (i.e., $\mathbf{x}^{(l-1)}$) and $\delta^{(l)}$. Subsequently, $\delta^{(l)}$ is used to calculate $\delta^{(l-1)}$. When $\partial e/\partial \mathbf{W}^{(l)}$ is computed for every layer, the weights are updated using the error gradients.

In the forward pass, the activations of the layer l, $\mathbf{x}^{(l)}$, cannot be computed until the activations of the previous layer, i.e., $\mathbf{x}^{(l-1)}$, are computed. In the backward pass, $\partial e/\partial \mathbf{W}^{(l)}$ can only be computed once, and $\mathbf{x}^{(l-1)}$ and $\delta^{(l)}$ have been computed. Moreover, $\delta^{(l)}$ depends on $\delta^{(l+1)}$. Finally, for a given minibatch, the backward pass cannot be started until the forward pass is completed and the error e has been determined.

The above dependences ensure that the weights of the layers are updated using the activations and error gradients calculated from the same batch of training data in one iteration of the backpropagation algorithm. Only when the weights are updated can the next batch of training data be fed into the network. These dependences limit parallelism when a network is partitioned across multiple accelerators and allow only one accelerator to be active at any point. This results in underutilization of the accelerators. It is this limitation that pipelining addresses.

3. Literature Review

There has been considerable work that explores parallelism in the training of deep networks. In data parallelism [14–19], each accelerator has a copy of the model. The accelerators process different minibatches of training data simultaneously in iterations, aggregating gradients to update weights at the end of each iteration. This is done synchronously [14, 17] or asynchronously [16]. More related to our work is model parallelism [16, 20–23] in which a large model is partitioned into different accelerators, each responsible for updating the weights for its portion of the model. The data dependences, described in Section 2, allow only one accelerator at a time to be active, resulting in underutilization. Pipelined parallelism addresses this underutilization and is the focus of our work. Below, we review salient work on pipelined parallelism in training.

Early work on pipelined training focuses on small networks and does not study pipelined parallelism in detail. Petrowski et al. [24] introduced the idea of pipelined backpropagation in neural network training. However, they realized the idea for only a 3-layer perceptron on a torus of 16 processors. Mostafa et al. [13] implemented a proof-of-concept validation of pipelined backpropagation training for a 3-layer fully connected binary-state neural network with truncated-error FPGA. However, the implementation does not have the coarse-grained layer-wise pipelined parallelization.

More recently, PipeDream [9] implemented pipelined training for large neural networks such as VGG-16, Inception-v3, and S2VT across multiple GPUs. It limits the usage of stale weights by a technique referred to as weight stashing. The technique keeps multiple versions of the weights during training, to ensure that the correct (i.e., nonstale) weights are used in each pipeline stage. This technique results in high inference accuracies and high utilization of the accelerators but increases the memory footprint of training.

GPipe [8] implements a library in TensorFlow to enable pipelined parallelism for the training of large neural networks. It pipelines microbatches within each minibatch to keep the gradients consistently accumulated. This eliminates the use of stale weight during training but at the expense of "pipeline bubbles" that degrade performance. GPipe utilizes these bubbles to reduce memory footprint by recomputing forward activations instead of storing them during the backward pass of training. The approach results in high inference accuracies with no increase in memory footprint, but the pipeline bubbles underutilize the accelerators, resulting in lower performance.

Huo et al. [12] implemented decoupled backpropagation (DDG) using delayed gradient updates. They showed that DDG guarantees convergence through a convergence analysis. Similar to PipeDream, DDG uses multiple copies of the weights, thus increasing memory footprint. Further, DDG pipelines only the backward pass of training, leaving forward pass unpipelined, which underutilizes resources. Huo et al. [25] followed up by proposing feature replay (FR) that recomputes activations during backward pass, similar to GPipe, reducing memory footprint and improving inference accuracy over DDG. Nonetheless, also similar to GPipe, the recomputations lower speedups.

Chen et al. [11] introduced weight prediction to mitigate weight staleness. Although their pipelined training shows improvement in throughput, they trained their networks for only 5000 iterations and it is not clear if their method can achieve standard model quality; their resulting model accuracies are much lower than typical for the models they train.

Guan et al. [26] presented XPipe, which combines elements of GPipe and PipeDream implementations of pipelined training to improve efficiency by allowing the overlapping of the pipelines of multiple microbatches from different minibatches. Nonetheless, they avoid the use of stale weights using weight prediction.

Kosson et al. [27] extended weight prediction in a fine-grained pipelined scheme that inserts pipeline registers between every pair of layers and limits the minibatch size to 1, aiming for a hardware implementation. They used a weight adjustment scheme to tackle weight staleness.

Park et al. [28] described HetPipe that combines data parallelism in the form of virtual workers with the pipelined parallelism of PipeDream, targeting heterogeneous clusters of GPU workstations. Jia et al. [29] proposed FlexFlow, a framework that explores data and model parallelism in the training, but they did not consider pipelined parallelism. Li et al. [30] proposed Pipe-SGD that pipelines computation and communication as opposed to the forward and backward passes. The model is not partitioned across the accelerators. Instead, the pipelining is used to overlap communication of weight updates and compute to hide

communication time and control the staleness at only 1 cycle. Therefore, large models may not fit on an accelerator.

A common theme to the above body of work is that it employs various techniques to avoid the use of stale weights. These techniques introduce either computational inefficiencies or memory footprint increases. In this work, we propose the use of stale weights and study their impact on the quality of trained models. We show that when pipelining is implemented in the early network stages or when hybrid training is used, we can train models with high prediction accuracy, smaller memory footprint, and higher performance.

For example, in contrast to PipeDream and DDG, we do not maintain multiple copies of weights, reducing memory footprint. In contrast to GPipe and Huo et al. [25], our approach has no pipeline bubbles and does not replicate computations, resulting in better performance. Further, compared to Chen et al. [11], our pipelined training can produce models with a final quality that is comparable to the standard model quality for VGG-16 and ResNet with different depths on CIFAR-10/CIFAR-100 datasets.

4. Proposed Pipelined Training Method

4.1. Pipelined Backpropagation. We illustrate our pipelined backpropagation implementation with the L layer network shown in Figure 3, using conceptual pipeline registers. Two registers are inserted between layers l and $l+1$, one register for the forward pass and a second for the backward pass. The forward register stores the activations of layer l ($\mathbf{x}^{(l)}$). The backward register stores the gradients $\delta^{(l+1)}$ of layer $l+1$. This defines a 4-stage pipelined backpropagation. The forward pass for layers 1 to l forms forward stage FS_1. The forward pass for layers $l+1$ to L forms forward stage FS_2. Similarly, the backward pass for layers $l+1$ to L and 1 to l forms backward stages BKS_1 and BKS_2, respectively.

The forward and backward stages are executed in a pipelined fashion on 3 accelerators: one for FS_1, one for both FS_2 and BKS_1, and one for BKS_2 (we combine FS_2 and BKS_1 on the same accelerator to reduce weight staleness, as will become evident shortly). In cycle 0, minibatch 0 is fed to FS_1. The computations of the forward pass are done as in the traditional nonpipelined implementation. In cycle 1, layer l activations $\mathbf{x}^{(l)}$ are fed to FS_2 and minibatch 1 is fed to FS_1. In cycle 2, the error for minibatch 0 computed in FS_2 is directly fed to BKS_1, the activations of layer $l\mathbf{x}^{(l)}$ are forwarded to FS_2, and minibatch 2 is fed to FS_1. This pipelined execution is illustrated by the space-time diagram in Figure 4 for 5 minibatches. The figure depicts the minibatch processed by accelerator cycles 0 to 6. At steady state, all the accelerators are active in each cycle of execution.

The above pipelining scheme utilizes weights in FS_1 that are yet to be updated by the errors calculated by FS_2 and BKS_1. At steady state, the activations of a minibatch in FS_1 are calculated using weights that are 2 execution cycles old or 2 cycles stale. This is reflected in Figure 4 by indicating the weights used by each forward stage and the weights updated by each backward stage. The weights of a forward stage are subscripted by how stale they are (negative subscripts). Similarly, the weights updated by a backward stage are subscripted by how delayed they are (positive subscripts).

Further, since the updates of the weights by BKS_2 require activations calculated for the same minibatch in FS_1 for all layers in the stage, it is necessary to save these activations until the error gradients with respect to the weights are calculated by BKS_2. Only when the weights are updated using the gradients can these activations be discarded.

In the general case, we use K pairs of pipeline registers (each pair consisting of a forward register and a backward register) inserted between the layers of the network. We describe the placement of the register pairs by the pipeline placement vector, $\mathrm{PPV} = (p_1, p_2, \ldots, p_K)$, where p_i represents the layer number after which a pipeline register pair is inserted. Such a placement creates $(K+1)$ forward stages, labeled $\mathrm{FS}_i, i = 1, 2, \ldots, K+1$, and $(K+1)$ backward stages, labeled $\mathrm{BKS}_i, i = 1, 2, \ldots, K+1$. Forward stage FS_i and backward stage BKS_{K-i+2} correspond to the same set of layers. Specifically, stage FS_i contains layers $p_i + 1$ to p_{i+1}, which are inclusive. We assign each forward stage and each backward stage to an accelerator, with the exception of the FS_{K+1} and backward stage BKS_1, which are assigned to the same accelerator to reduce weight staleness by an execution cycle. In total, $2K+1$ accelerators are used.

We quantify weight staleness as follows. A forward stage FS_i and backward stage BKS_{K-i+2} use the same weights that are $2(K-i+1)$ cycles old. Further, a forward stage FS_i must store the activations of all layers in the stage for all $2(K-i+1)$ cycles which are used for the corresponding backward stage BKS_{K-i+2}. We refer to these saved activations as intermediate activations. We define the degree of staleness as $2(K-i+1)$. For each pair of stages FS_i and BKS_{K-i+2}, let there be N_i weights in their corresponding layers. The layers before the last pipeline register pairs always use stale weights. Thus, we define percentage of stale weight as $(\sum_{i=1}^{K} N_i)/(\sum_{i=1}^{K+1} N_i)$.

On the one hand, the above pipelined execution allows a potential speedup of $2K+1$, using as many accelerators, over the nonpipelined implementation, keeping all the accelerators active at steady state. On the other hand, the use of stale weights may prevent training convergence or may result in a model that has an inferior inference accuracy. Further, it requires an increase in storage for activations. Our goal is to assess the benefit of this pipelined execution and the impact of its downsides.

Appendix C presents an analytical proof of the convergence of our pipelined training scheme.

4.2. Hybrid Pipelined/Nonpipelined Backpropagation. Hybrid training combines pipelined training with nonpipelined training. We start with pipelined training and after a number of iterations, we switch to nonpipelined training. This can address drops in inference accuracy of resulting models because of weight staleness, but it reduces the performance benefit since during nonpipelined training, the accelerators are underutilized.

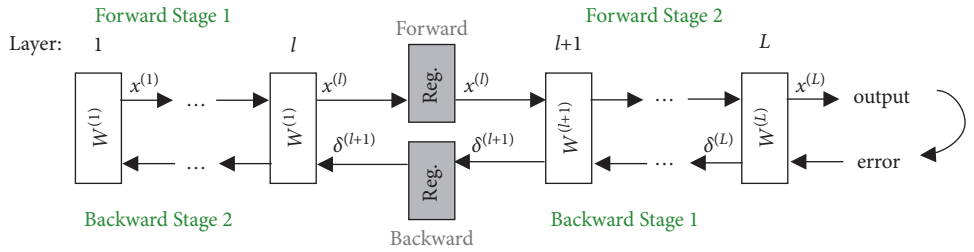

Figure 3: Pipelined backpropagation algorithm.

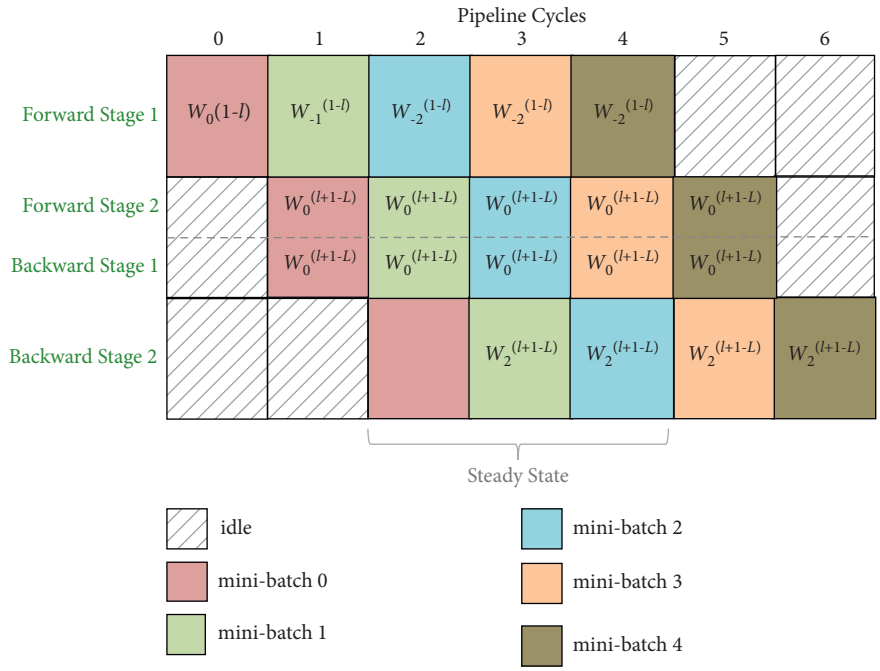

Figure 4: Illustration of pipelined computations of each cycle.

The extent of the speedup obtained by hybrid training with a given number of accelerators is determined by the number of iterations used for pipelined and nonpipelined training. Assume that n_{np} iterations are used to reach the best inference accuracy for nonpipelined training, and that in hybrid training, n_p iterations ($n_p \leq n_{np}$) are pipelined followed by $n_{np} - n_p$ iterations of nonpipelined training to reach the same inference accuracy as nonpipelined training. The speedup of hybrid training with respect to the nonpipelined training with $2K + 1$ accelerators is $n_{np}/(n_p/(2K+1) + (n_{np} - n_p))$. For large K, the speedup approaches an upper bound of $n_{np}/(n_{np} - n_p)$.

5. Implementation

We implement pipelined training in two ways: simulated in Caffe [31] (version 1.0.0), where the whole training process is performed on one process with no parallelism, and actual with parallelism across accelerators in PyTorch [32] (version 1.0.0.dev20190327).

The simulated implementation is used to analyze statistical convergence, inference accuracy, and impact of weight staleness, for a large number of stages/accelerators, unconstrained by parallelism and communication overhead. In contrast, the actual implementation reports real performance and serves as a proof-of-concept implementation that demonstrates the performance potential of pipelined training with stale weights. PyTorch is used instead of Caffe to leverage its support for collective communication protocols and its flexibility in partitioning a network across multiple accelerators. The versions of Caffe and PyTorch we use have no support for pipelined training. Thus, both were extended to provide such support.

We develop a custom Caffe layer in Python, which we call a Pipeline Manager Layer (PML), to facilitate the simulated pipelining. During the forward pass, a PML registers the input from a previous layer and passes the activation to the next layer. It also saves the activations for the layers connected to it to be used in the backward pass. During the backward pass, a PML passes the appropriate error gradients. It uses the corresponding activations saved during the forward pass to update weights and generate error gradients for the previous stage, using existing weight update mechanisms in Caffe.

To implement actual hardware-accelerated pipelined training, we partition the network onto different accelerators (GPUs), each running its own process. Activation and gradient data are communicated among accelerators using an asynchronous send/receive communication protocol, but all communication must go through the host CPU, since point-to-point communication between accelerators is not supported in PyTorch. This increases communication overhead. Similar to the PMLs in Caffe, the activations computed on one GPU are copied to the next GPU (via the CPU) in the forward pass and the error gradients are sent (again via the CPU) to the preceding GPU during the backward pass. The GPUs are running concurrently, achieving pipeline parallelism.

6. Evaluation

6.1. Setup, Methodology, and Metrics. Simulated pipelining is evaluated on a machine with one Nvidia GTX1060 GPU with 6 GB of memory and an Intel i9-7940X CPU with 64 GB of RAM. The performance of actual pipelining is evaluated using two Nvidia GTX1060 GPUs, each with 6 GB of memory, hosted in an Intel i7-9700K machine with 32 GB of RAM.

We use a number of CNNs and datasets in our evaluation. For the simulated implementation, we use LeNet-5 [33] trained on MNIST [34], AlexNet [35], VGG-16 [36], and ResNet [1], all trained on CIFAR-10 [37]. We evaluate simulated pipelining with these networks/datasets to manage simulation time. For the actual implementation, we experiment with progressively larger ResNet depths: 56, 110, 224, and 362, trained on CIFAR-10 and CIFAR-100 [38]. In both the simulated and actual implementations, we train the networks following their original setting [1, 33, 35, 36], with minor variations to the hyperparameters, as described in Appendix A and Appendix B.

We elect to use the above CNNs for two reasons. First, they are commonly used in the evaluation of pipelined training (e.g., VGG in PipeDream [9] and ResNet in GPipe [8], which we compare to in our evaluation). Second, these networks have increasing sizes, ranging from the small LeNet to the large VGG and the progressively larger ResNets. This range in size allows us to effectively assess the impact of stale weight on pipelined training. We leave the use of larger networks, such as BERT [39] or DLRM [40] to future work.

We evaluate the effectiveness of pipelined training in terms of its training convergence and its Top-1 inference accuracy, compared to those of the nonpipelined training. We use the speedup to evaluate performance improvements. The speedup is defined as the ratio of the training time of the nonpipelined implementation on single communication-free GPU to the training time of the pipelined training.

6.2. Training Convergence and Inference Accuracy. Pipelined training is done using 4, 6, 8, and 10 pipeline stages. Table 1 shows where the registers are inserted in the networks using their PPVs (defined in Section 4). Pipeline registers are inserted among groups of convolutional layers, resulting up to 8 pipeline stages for AlexNet and ResNet-20 and 10 pipeline stages for LeNet-5 and VGG-16.

Table 1: Pipeline placement vectors for CNNs.

CNN	Number of layers	4-Stage	6-Stage	8-Stage	10-Stage
LeNet-5	5	(1)	(1, 2)	(1, 2, 3)	(1, 2, 3, 4)
AlexNet	8	(1)	(1, 2)	(1, 2, 3)	—
VGG-16	16	(2)	(2, 4)	(2, 4, 7)	(2, 4, 7, 10)
ResNet-20	20	(7)	(7, 13)	(7, 13, 19)	—

Figure 5 shows the improvements in the inference accuracies for both pipelined and nonpipelined training as a function of the number of training iterations (each iteration corresponds to a minibatch). The figure shows that for all the networks, both pipelined training and nonpipelined training have similar convergence patterns. They converge in more or less the same number of iterations for a given number of pipeline stages, albeit different inference accuracies. This indicates that our approach to pipelined training with stale weights does converge, similar to nonpipelined training.

Table 2 shows the inference accuracies obtained after up to 30,000 iterations of training. For LeNet-5, the inference accuracy drop is within 0.5%. However, for the other networks, there is a small drop in inference accuracy with 4 and 6 stages. AlexNet has about 4% drop in inference accuracy, but for VGG-16, the inference accuracy drop is within 2.4%, and for ResNet-20, the accuracy drop is within 3.5%. Thus, the resulting model quality is generally comparable to that of a nonpipelining-trained model.

However, with deeper pipelining (i.e., 8 and 10 stages), inference accuracies significantly drop. There is a 12% and a 8.5% inference accuracy drop for VGG-16 and ResNet-20, respectively. In this case, the model quality is not comparable to that of the nonpipelined training. This result confirms what is reported in the literature [9] and is attributed to the use of stale weights.

6.3. Impact of Weight Staleness. We wish to better understand the impact of the number of pipeline stages and their location in the network on inference accuracy. We focus on ResNet-20 because of its relatively small size and regular structure. It consists of 3 residual function groups with 3 residual function blocks within each group. In spite of this relatively small size and regular structure, it enables us to create pipelines with up to 20 stages by inserting pipeline register pairs within residual function blocks.

We conduct two experiments. In the first, we increase the number of pipeline stages (from earlier layers to latter layers) and measure the inference accuracy of the resulting model. The results are shown in Table 3, which gives the inference accuracy of pipelined training after 100,000 iterations, as the number of pipeline stages increases. The 8-stage pipelined training is created by a PPV of (3, 5, 7), and the subsequent pipeline schemes are created by adding pipeline registers after every 2 layers after layer 7. Clearly, the greater the number of stages is, the worse the resulting model quality is.

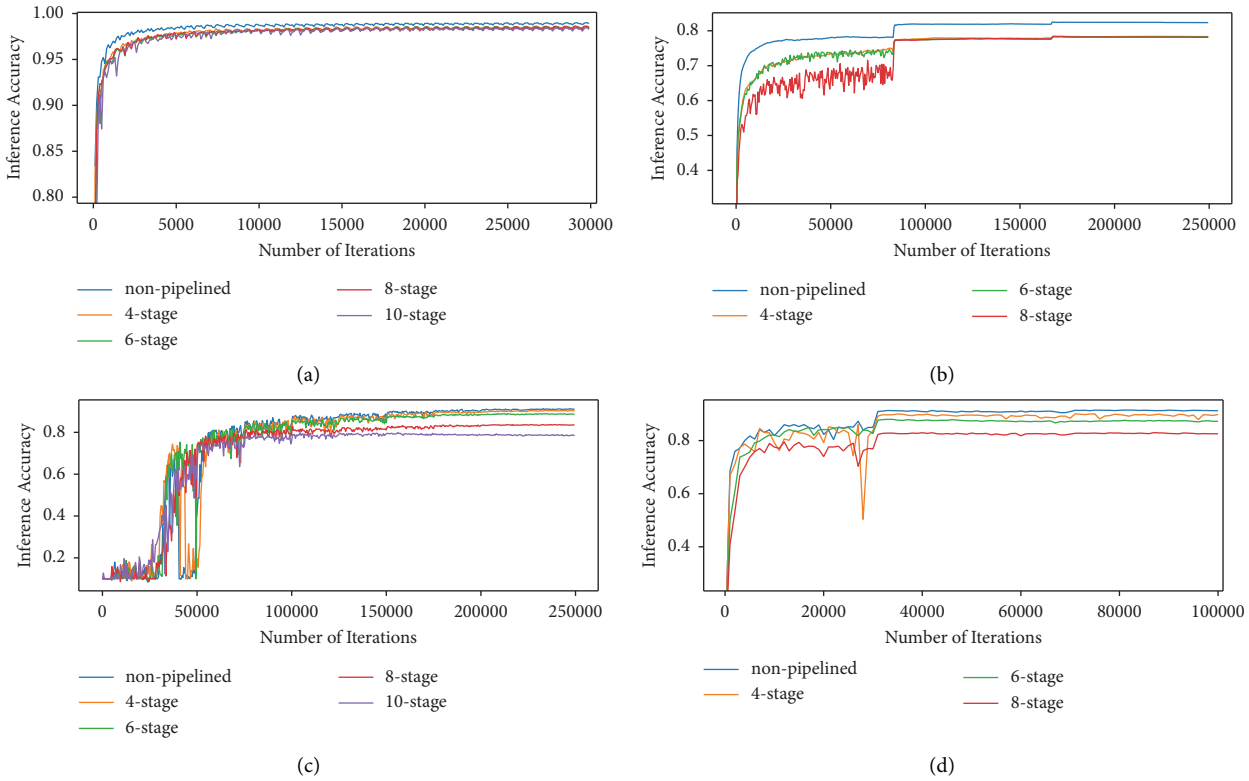

Figure 5: Inference accuracy curves for simulated pipelined training. (a) LeNet-5 on MNIST. (b) AlexNet on CIFAR-10. (c) VGG-16 on CIFAR-10. (d) ResNet-20 on CIFAR-10.

Table 2: Inference accuracy for simulated pipelined training.

CNN	Nonpipelined (%)	4-Stage (%)	6-Stage (%)	8-Stage (%)	10-Stage (%)
LeNet-5	99.00	98.64	98.62	98.61	98.47
AlexNet	82.51	78.47	78.32	78.47	—
VGG-16	91.36	90.53	88.96	83.73	79.85
ResNet-20	91.50	90.05	88.00	83.01	—

Table 3: Fine-grained pipelining inference accuracy.

Stages	Inference accuracy (%)
Nonpipelined	91.50
8	90.28
10	88.37
12	88.73
14	87.94
16	87.30
18	86.23
20	79.09

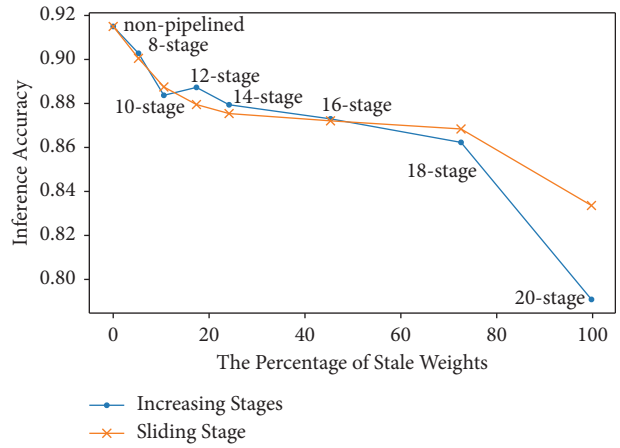

Figure 6: Inference accuracy vs % of stale weights.

The number of stale weights used in the pipelined training increases as the number of pipeline stages increases. Thus, Figure 6 depicts the inference accuracy as a function of the percentage of weights that are stale. The curve labeled "increasing stages" shows that the drop in inference accuracy increases as the percentage of stale weights increases.

In the second experiment, we investigate the impact of the degree of staleness described in Section 4. Only one pair of pipeline registers is inserted. The position of this register slides from the beginning of the network to its end. At every position, the percentage of stale weights remains the same as in the first experiment, but all stale weights have the same

degree of staleness. The result of this experiment is shown by the curve labeled "sliding stage" in Figure 6. The curve shows the inference accuracy also drops as the percentage of stale weights increases. However, it also indicates that the drop of inference accuracy remains more or less the same as in the first experiment in which the degree of staleness is higher. Thus, the percentage of stale weight appears to be what determines the drop in inference accuracy and not the degree of staleness of the weights.

The percentage of stale weights is determined by where the last pair of pipeline registers are placed in the network. It is the position of this pair that determines the loss in inference accuracy. Therefore, it is desirable to place this last pair of registers as early as possible in the network so as to minimize the drop in inference accuracy.

While at first glance this may seem to limit pipelining, it is important to note that the bulk of computations in a CNN is in the first few convolutional layers in the network. Inserting pipeline registers for these early layers can result in a large number of stages that are computationally balanced. For example, our profiling of the runtime of ResNet-20 shows that the first three residual functions take more than 50% of the training runtime. This favors more pipeline stages at the beginning of the network. Such placement has the desirable effect of reducing the drop in inference accuracy while obtaining relatively computationally balanced pipeline stages.

6.4. Effectiveness of Hybrid Training. We demonstrate the effectiveness of hybrid training, also using ResNet-20. Figure 7 shows the inference accuracy for 20 K iterations of pipelined training followed by either 10 K or 20 K iterations of nonpipelined training. This inference accuracy is compared to 30 K iterations of either nonpipelined or pipelined training with PPV (5, 12, 17). The figure demonstrates that hybrid training converges in a similar manner to both pipelined and nonpipelined training. Table 4 shows the resulting inference accuracies. The table shows that the 20 K + 10 K hybrid training produces a model with accuracy that is comparable to that of the nonpipelined model. Further, with an additional 10 K iterations of nonpipelined training, the model quality is slightly better than that of the nonpipelined model. This demonstrates the effectiveness of hybrid training.

6.5. Pipelined and Hybrid Training Performance. Our evaluation using simulated pipelining explored pipelines with up to 20 pipeline stages (up to 10 accelerators). In this section, we implement and evaluate a proof-of-concept implementation with actual pipelining. The goal is to demonstrate that pipelined training with stale weights, with and without hybrid training, does deliver performance improvements.

Specifically, we implement 4-stage pipelined training for ResNet-56/110/224/362 on a 2-GPU system. Each GPU is responsible for one forward stage and one backward stage. Thus, the maximum speedup that can be obtained is 2. We train every ResNet for 200 epochs for CIFAR-10 dataset and 300 epochs for CIFAR-100 dataset. Tables 5 and 6 show the inference accuracies with and without pipelining, for the CIFAR-10 and CIFAR-100 datasets. They also show the speedups of pipelined training over the nonpipelined one. The tables indicate that the quality of the models produced by pipelined training is comparable to those achieved by the simulated pipelining implementation. The tables also show that speedup exists for all networks. Indeed, for ResNet-362, the speedup is 1.8X. This is equivalent to about 90% utilization for each GPU. Finally, the tables reflect that as the networks get larger, the speedup improves. This is because for larger networks, the ratio of computation to communication overhead is higher, leading to better speedups.

Moreover, we combine the 4-stage pipelined training described above with nonpipelined training to demonstrate the performance of hybrid training. We train every ResNet using pipelined training for 100 epochs and 150 epochs and follow it up by 100 epochs and 150 epochs of nonpipelined training for CIFAR-10 and CIFAR-100, respectively. Because the maximum speedup for the pipelined training is 2 and only half the training epochs is accelerated, the maximum speedup for this hybrid training is $s = t/(t/2 + t/4) = 1.33$, where t is the training time of nonpipelined training. Tables 5 and 6 also show the inference accuracies and speedup of the hybrid training for each ResNet and validate that hybrid training can produce a model quality that is comparable to the baseline nonpipelined training while speeding up the training process. Indeed, hybrid training can sometimes produce models with slightly better inference accuracies. Similar to pipelined training, as network size grows, the speedup of hybrid training reaches 1.29X, approaching the theoretical limit of 1.33X.

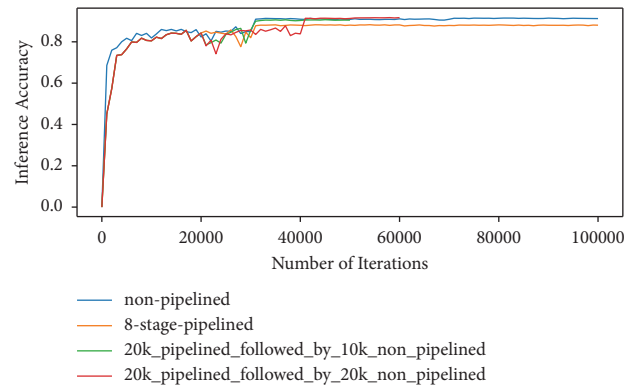

Figure 7: Hybrid training convergence.

Table 4: Hybrid training inference accuracy.

	Inference accuracy (%)
Baseline (30 K)	91.50
Pipelined (30 K)	88.29
20 K + 10 K hybrid	90.71
20 K + 20 K hybrid	91.72

Table 5: Inference accuracy and speedup of actual pipelined/hybrid training for CIFAR-10.

ResNet	PPV	Accuracy			Time (seconds)			Speedup	
		Nonpipelined (%)	Pipelined (%)	Hybrid (%)	Nonpipelined	Pipelined	Hybrid	Pipelined	Hybrid
-56	(19)	92.63	92.89	92.75	6,745	4,090	5,429	1.65X	1.24X
-110	(37)	93.59	92.88	93.55	13,150	7,570	10,452	1.73X	1.26X
-224	(75)	92.77	91.39	93.33	27,231	14,998	21,245	1.81X	1.28X
-362	(121)	93.46	90.53	93.98	44,814	24,640	34,814	1.82X	1.29X

Table 6: Inference accuracy and speedup of actual pipelined/hybrid training for CIFAR-100.

ResNet	PPV	Accuracy			Time (seconds)			Speedup	
		Nonpipelined (%)	Pipelined (%)	Hybrid (%)	Nonpipelined	Pipelined	Hybrid	Pipelined	Hybrid
-56	(19)	72.30	70.25	72.34	10,068	6,040	8,113	1.67X	1.24X
-110	(37)	72.63	71.52	73.03	19,624	11,162	15,591	1.76X	1.26X
-224	(75)	73.00	70.90	72.69	40,807	22,561	32,132	1.81X	1.28X
-362	(121)	71.77	71.03	72.09	66,977	37,008	51,688	1.81X	1.29X

Table 7: Memory usage of 4-stage pipelined ResNet training.

CNN	PPV	Activations	Weights (MB)	Increase	Increase percentage in upper bound
ResNet-56	(19)	10.87 MB × batch size	3.25	6.32 MB × batch size	58
ResNet-110	(37)	21.43 MB × batch size	6.59	12.35 MB × batch size	57
ResNet-224	(75)	43.70 MB × batch size	13.64	25.07 MB × batch size	57
ResNet-362	(121)	70.67 MB × batch size	22.17	40.50 MB × batch size	57

Table 8: Learning rate of BKS_2.

	BKS_2 learning Rate
ResNet-56	0.01
ResNet-110	0.001
ResNet-224	0.001
ResNet-362	0.001

6.6. Memory Usage. Pipelined training requires the saving of intermediate activations, as described earlier in Section 4, leading to an increase in memory footprint. This increase in memory is a function of not only the placement of the pipeline registers but also of the network architecture, input size, and the minibatch size. We calculate the memory usage of the 4-stage pipelined ResNet training above to show that this increase is modest for our pipelining scheme. Specifically, we use torchsummary in PyTorch to report memory usage for weights and activations for a network and calculate the additional memory required by the additional copies of activations. Assuming a batch size of 128, the percentage increase in memory usage is no more than 58% for ResNet-56/110/224/362 (see Table 7). More analysis of memory increase appears in Appendix D.

6.7. Comparison to Existing Work. We compare our pipelined training scheme with two key existing systems: PipeDream [9] and GPipe [8]. We believe that PipeDream and GPipe are representative of existing key approaches that implement pipelined training, including decoupled backpropagation (DDG) [12] and feature replay (FR) [25] (discussed in Section 3). We compare on the basis of three aspects: the pipelining scheme, performance, and memory usage.

Our pipelining scheme is simpler than that of PipeDream and GPipe in that we do not require weight stashing nor do we divide minibatches into microbatches. This leads to less communication overhead and is amicable to rapid realization in machine learning framework such as PyTorch or in actual hardware such as Xilinx's xDNN FPGA accelerators [41].

Our pipelining scheme, as PipeDream, eliminates bubbles that exist in the pipeline leading to better performance. For example, we obtain a speedup of 1.7X for ResNet-110 using 2 GPUs in contrast to GPipe that obtains a speedup of roughly 1.3X for ResNet-101 using 2 TPUs. We also obtain similar performance compared to PipeDream for similar networks. When the number of pipeline stages grows, pipeline bubbles exhibit more negative effect on performance shown in GPipe on a 4-partition pipelined ResNet-101 using 4 TPUs as its bubble overhead doubled compared to that of the 2-partition pipelined ResNet-101.

Our scheme uses less memory compared to PipeDream, although it introduces more memory overhead compared to GPipe. PipeDream saves intermediate activations during

Initialize weights $w^0 = [w^0_{p(1)}, \ldots, w^0_{p(k)}] \in \mathbb{R}^d$
Given learning rate sequence $\{\eta_t\}$
for $t = 0, 1, 2$ **to** $T - 1$ **do**
  for $k = 0, 1, 2$ **to** K **in parallel do**
    Compute gradient using stale weights:
      $g^t_k \longleftarrow [\nabla f_{p(k), x_{i(t-K+k)}}(w^t)]_{p(k)}$
    Update weights:
      $w^{t+1}_{p(k)} \longleftarrow w^t_{p(k)} - \eta_t \cdot g^t_k$
  end for
end for

Algorithm 1: Pipelined stochastic gradient descent (PPL-SGD).

Table 9: LeNet-5, AlexNet, VGG-16, and ResNet-20 memory increase of 4-stage pipelined training.

CNN	Dataset	PPV	Activation memory for minibatch size 1 (MB)	Minibatch size	Total weight memory (MB)	Memory increase % of PipeDream	Memory increase % of this work
LeNet-5	MNIST	(2)	0.06	128	0.24	109	97
AlexNet	CIFAR-10	(3)	0.88	128	88.87	214	37
VGG-16	CIFAR-10	(2)	3.30	128	58.16	124	75
ResNet-20	CIFAR-10	(7)	3.84	128	1.03	61	60
VGG-16	ImageNet	(2)	218.59	32	527.79	105	77

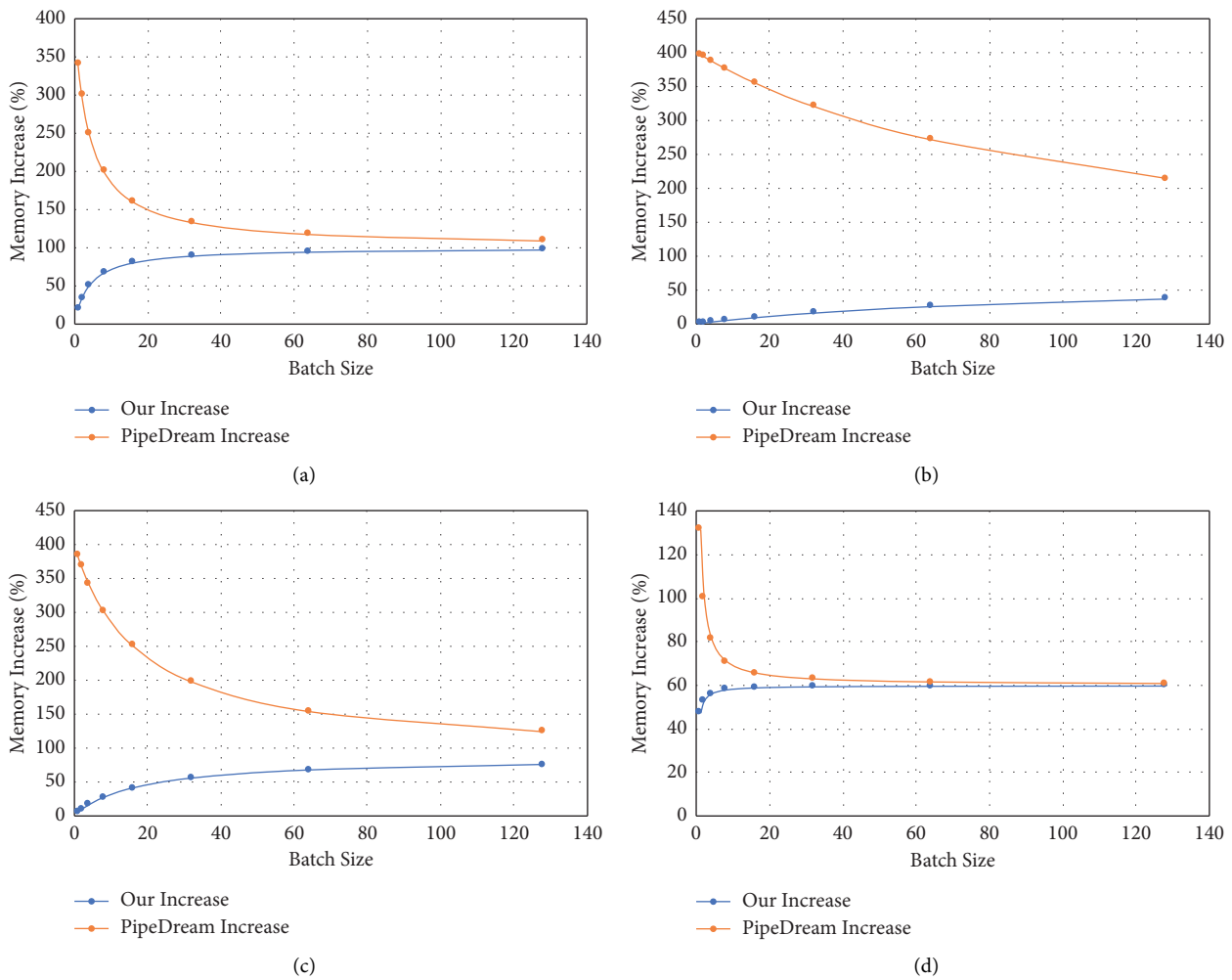

Figure 8: Memory increase vs batch size for 4-stage pipelined training. (a) LeNet-5 on MNIST. (b) AlexNet on CIFAR-10. (c) VGG-16 on CIFAR-10. (d) ResNet-20 on CIFAR-10.

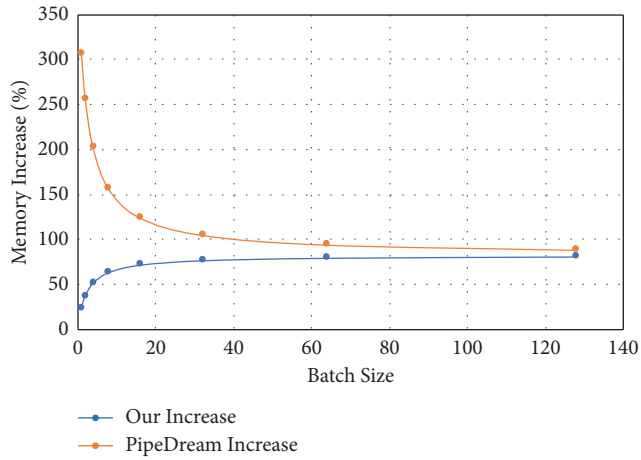

Figure 9: Memory increase vs batch size for 4-stage pipelined training for VGG-16 on ImageNet.

training, as we do. However, it also saves multiple copies of a network's weights for weight stashing. The memory footprint increase due to weight stashing depends on the total weight memory compared to activation memory, the number of active minibatches in the training pipeline, the minibatch size, and the training dataset. In some cases, weight stashing can have a significant impact on memory footprint. For example, for AlexNet trained on CIFAR-10 with a minibatch size of 128 using a 4-stage pipelined training, in which the weight memory is much larger than the activation memory, PipeDream's memory footprint increase is 177% more than ours. A more detailed memory usage comparison is presented in Appendix D.

7. Concluding Remarks

We propose and evaluate a pipelined execution scheme of backpropagation for the training of CNNs. The scheme uses stale weights, fully utilizes accelerators, does not significantly increase memory usage, and results in models with comparable prediction accuracies to those obtained with nonpipelined training.

The use of stale weights has been recognized in the literature to significantly affect prediction accuracies. Thus, existing schemes avoid or limit the use of stale weights [7–9, 12]. In contrast, we explore the impact of stale weights and demonstrate that it is the placement of the last pair of pipeline registers that determines the loss in inference accuracy. This allows us to implement pipelining in the early layers of the network with little loss to accuracy while reaping computational benefits. Limiting pipelining to such early layers is not a disadvantage since the bulk of computations is in the early convolutional layers. Nonetheless, when deeper pipelining is desired, we introduce hybrid training and show that it is effective in mitigating the loss of prediction accuracy for deep pipelining, while still providing computational speedups. Our scheme has the advantage of simplicity and low memory overhead, making it attractive when accelerator memory is constrained, in particular for specialized hardware accelerators.

Our evaluation using several CNN networks/datasets confirms that training with our scheme does converge and does produce models with inference accuracies that are comparable to those obtained with nonpipelined training. Our proof-of-concept implementation on a 2-GPU system shows that our scheme achieves a speedup of up to 1.82X, demonstrating its potential.

This work can be extended in a number of directions. One direction is to evaluate the approach with a larger number of accelerators since pipelined parallelism is known to scale naturally with the number of accelerators. Another is to evaluate the approach on larger datasets, such as ImageNet. Finally, our pipelining scheme lends itself naturally to hardware implementation due to its simplicity. Thus, another direction for future work is to evaluate pipelined parallelism using Field Programmable Gate Array (FPGA) or ASIC accelerators.

Appendix

A. Training Hyperparameters for Simulated Training

LeNet-5 is trained on the MNIST dataset with stochastic gradient descent (SGD) using a learning rate of 0.01 with inverse learning policy, a momentum of 0.9, a weight decay of 0.0005, and a minibatch size of 100 for 30,000 iterations. The progression of inference accuracy during training is recorded with 300 tests.

AlexNet is trained on the CIFAR-10 dataset with SGD with Nesterov momentum using a learning rate of 0.001 that is decreased by 10X twice during training, a momentum of 0.9, a weight decay of 0.004, and a minibatch size of 100 for 250,000 iterations. One test is performed every epoch to record the progression of inference accuracy.

VGG-16 is trained on CIFAR-10 dataset with SGD with Nesterov momentum using a learning rate starting at 0.1 that is decreased by half every 50 epochs during training, a momentum of 0.9, a weight decay of 0.0005,

and a minibatch size of 100 for 250,000. Since it is relatively more difficult to train VGG-16 compared to other models, batch normalization and dropout are used during training throughout the network. One test is performed every epoch to record the progression of inference accuracy.

ResNet is trained on CIFAR-10 dataset with SGD using a learning rate starting at 0.1 and 0.01 for nonpipelined and pipelined training, respectively, that is decreased by 10X twice during training, a momentum of 0.9, a weight decay of 0.0001, and a minibatch size of 128 for 100,000 iterations. Batch normalization is used during training throughout the network. One test is performed every 100 iterations to record the progression of inference accuracy.

B. Training Hyperparameters for Actual Training

For the baseline nonpipelined training, ResNet-56/110/224/362 is trained on CIFAR-10 and CIFAR-100 dataset for 200 and 300 epochs, respectively, with SGD using a learning rate of 0.1 that is decreased by a factor of 10 twice (at epoch 100 and 150 for CIFAR-10 and at epoch 150 and 225 for CIFAR-100), a momentum of 0.9, a weight decay of 0.0001, and a minibatch size of 128. Batch normalization is used during training throughout the network. This set of hyperparameters can be found at https://github.com/akamaster/pytorch_resnet_cifar10.

For the 4-stage pipelined training, the hyperparameters are the same as the nonpipelined baseline, except for the BKS_2 learning rate. Table 8 shows the learning rate for all ResNet experimented.

C. Convergence Analysis

Experimental evaluation shows that our pipelined training converges for large networks. Nonetheless, a convergence analysis provides a theoretical foundation for our pipelined training across networks. Our analysis is analogous to that of Bottou et al. [42] and Huo et al. [12] in that it shows that our pipelined training algorithm has similar convergence rate to both decoupled parallel backpropagation and nonpipelined stochastic gradient descent. Our training algorithm is summarized in Algorithm 1. We show that this algorithm converges in a fashion similar to Huo et al. [12].

We start by making the same assumption as in [12, 42]. Specifically we make the Lipschitz-continuous gradient assumption that guarantees that $\|\nabla f(u) - \nabla f(v)\|_2 \leq L\|u - v\|_2$. In this assumption, $f(.)$ is the error function, $L > 0$, and $u, v \in \mathbb{R}^d$. We also make the bounded variance assumption that guarantees that $\|\nabla f_{x_i}(w)\|_2^2 \leq M$, where $f(.)$ is the error function, $M > 0$, for any sample x_i, and $\forall w \in \mathbb{R}^d$. Because of the unnoised stochastic gradient $\mathbb{E}[\nabla f_{x_i}(w)] = \nabla f(w)$ and $\mathbb{E}\|\nabla f_{x_i}(w) - \nabla f(w)\|_2^2 = \mathbb{E}\|\nabla f_{x_i}(w)\|_2^2 - \|\nabla f(w)\|_2^2$, the variance of the stochastic gradient is guaranteed to be less than M.

Based on these two assumptions, if there are K forward stages in our pipelined scheme, each iteration of Algorithm 1 satisfies the following inequality $\forall t \in \mathbb{N}$:

$$\mathbb{E}\left[f\left(w^{t+1}\right)\right] - f\left(w^t\right) \leq -\left(\eta_t - \frac{L\eta_t^2}{2}\right)\left\|\nabla f\left(w^t\right)\right\|_2^2 + L\eta_t^2 KM. \tag{C.1}$$

This can be shown true as follows. From the Lipschitz-continuous gradient assumption, we obtain the following inequality:

$$f\left(w^{t+1}\right) \leq f\left(w^t\right) + \nabla f\left(w^t\right)^T\left(w^{t+1} - w^t\right) + \frac{L}{2}\left\|w^{t+1} - w^t\right\|_2^2. \tag{C.2}$$

From the weight update rule in Algorithm 1, we take expectation on both sides of inequality 2 and obtain the following:

$$\begin{aligned}
\mathbb{E}\left[f\left(w^{t+1}\right)\right] &\leq f\left(w^t\right) + \eta_t \mathbb{E}\left[\nabla f\left(w^t\right)^T\right]\left(\sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}\left(w^t\right)\right) + \frac{\eta_t^2 L}{2}\mathbb{E}\left\|\sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}\left(w^t\right)\right\|_2^2, \\
&\leq f\left(w^t\right) - \eta_t \sum_{k=1}^{K} \nabla f\left(w^t\right)^T\left(\nabla f_{p(k)}\left(w^t\right)\right) + \nabla f_{p(k)}\left(w^t\right) - \nabla f_{p(k)}\left(w^t\right) + \frac{\eta_t^2 L}{2}\mathbb{E}\left\|\sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}\left(w^t\right) + \nabla f\left(w^t\right) - \nabla f\left(w^t\right)\right\|_2^2 \\
&= f\left(w^t\right) - \eta_t\left\|\nabla f\left(w^t\right)\right\|_2^2 - \eta_t \sum_{k=1}^{K} \nabla f\left(w^t\right)\left(\nabla f_{p(k)}\left(w^t\right) - \nabla f_{p(k)}\left(w^t\right)\right) + \frac{L\eta_t^2}{2}\left\|\nabla f\left(w^t\right)\right\|_2^2 \\
&\quad + \frac{L\eta_t^2}{2}\mathbb{E}\left\|\sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}\left(w^t\right) - \nabla f\left(w^t\right)\right\|_2^2 + L\eta_t^2 \sum_{k=1}^{K} \nabla f\left(w^t\right)^T\left(\nabla f_{p(k)}\left(w^t\right) - \nabla f_{p(k)}\left(w^t\right)\right) \\
&= f\left(w^t\right) - \left(\eta_t - \frac{L\eta_t^2}{2}\right)\left\|\nabla f\left(w^t\right)\right\|_2^2 + \frac{L\eta_t^2}{2}\mathbb{E}\left\|\sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}\left(w^t\right) - \nabla f\left(w^t\right)\right\|_2^2.
\end{aligned} \tag{C.3}$$

Let us define the last term in inequality (C.3) as $Q = (L\eta_t^2/2)\mathbb{E}\| \sum_{k=1}^{K} \nabla f_{p(k),x_{i(t-K+k)}}(w^t) - \nabla f(w^t)\|_2^2$. Because $\|x+y\|_2^2 \le 2\|x\|_2^2 + 2\|y\|_2^2$ and $\mathbb{E}\|x - \mathbb{E}[x]\|_2^2 = \mathbb{E}\|x\|_2^2 - \|\mathbb{E}[x]\|_2^2$, we can derive an upper bound for Q:

$$\begin{aligned} Q &= \frac{L\eta_t^2}{2}\mathbb{E}\left\|\sum_{k=1}^{K}\nabla f_{p(k),x_{i(t-K+k)}}(w^t) - \nabla f(w^t) - \sum_{k=1}^{K}\nabla f_{p(k)}(w^t) + \sum_{k=1}^{K}\nabla f_{p(k)}(w^t)\right\|_2^2, \\ &\le L\eta_t^2\mathbb{E}\left\|\sum_{k=1}^{K}\nabla f_{p(k),x_{i(t-K+k)}}(w^t) - \sum_{k=1}^{K}\nabla f_{p(k)}(w^t)\right\|_2^2 + L\eta_t^2\left\|\sum_{k=1}^{K}\nabla f_{p(k)}(w^t) - \nabla f(w^t)\right\|_2^2 \\ &= L\eta_t^2\sum_{k=1}^{K}\mathbb{E}\left\|\nabla f_{p(k),x_{i(t-K+k)}}(w^t) - \nabla f_{p(k)}(w^t)\right\|_2^2 + L\eta_t^2\sum_{k=1}^{K}\left\|\nabla f_{p(k)}(w^t) - \nabla f_{p(k)}(w^t)\right\|_2^2 \\ &\le L\eta_t^2\sum_{k=1}^{K}\mathbb{E}\left\|\nabla f_{p(k),x_{i(t-K+k)}}(w^t)\right\|_2^2 \le L\eta_t^2 KM. \end{aligned} \tag{C.4}$$

From inequalities (C.3) and (C.4), we have the following inequality:

$$\mathbb{E}\left[f\left(w^{t+1}\right)\right] - f\left(w^t\right) \le -\left(\eta_t - \frac{L\eta_t^2}{2}\right)\left\|\nabla f\left(w^t\right)\right\|_2^2 + L\eta_t^2 KM. \tag{C.5}$$

This proves inequality (C.1).

From inequality (C.1), if the value of learning η_t is picked such that the right-hand size of inequality (C.1) is less than zero, the error function is decreasing. Therefore, using this property, we can analyze the convergence of Algorithm 1 for a fixed learning rate and a decreasing learning rate.

For a fixed learning rate η, we show that Algorithm 1 converges. Given the Lipschitz-continuous gradient and the Bounded variance assumption and a fixed learning rate $\eta_t = \eta, \forall t \in \{0, 1, \ldots, T-1\}$ and $\eta L \le 1$, if we assume that the optimal solution that minimizes our error function $f(w)$ is w^*, then the output of our Algorithm 1 satisfies the following inequality:

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 \le \frac{f\left(w^0\right) - f\left(w^*\right)}{\left(\eta - L\eta^2/2\right)T} + \eta^2 LKM. \tag{C.6}$$

This inequality holds because when η_t is constant and $\eta_t = \eta$, taking expectation of inequality (C.1), we have

$$\mathbb{E}\left[f\left(w^{t+1}\right)\right] - \mathbb{E}\left[f\left(w^t\right)\right] \le -\left(\eta - \frac{L\eta^2}{2}\right)\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 + L\eta^2 KM. \tag{C.7}$$

Summing inequality (C.7) from $t = 0$ to $T-1$, we have

$$\mathbb{E}\left[f\left(w^T\right)\right] - f\left(w^0\right) \le -\left(\eta - \frac{L\eta^2}{2}\right)\sum_{t=0}^{T-1}\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 + TL\eta^2 KM. \tag{C.8}$$

Suppose that w^* is the optimal solution for $f(w)$; then, $f(w^*) - f(w^0) \le \mathbb{E}[f(w^T)] - f(w^0)$, and the following inequality is obtained:

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 \le \frac{f\left(w^0\right) - f\left(w^*\right)}{\left(\eta - L\eta^2/2\right)T} + \eta^2 LKM, \tag{C.9}$$

thus proving inequality (C.6).

In inequality (C.6), when $T \longrightarrow \infty$, the average norm of the error gradient is bounded by $\eta^2 LKM$ that is finite. This shows that Algorithm 1 converges for a fixed learning rate η.

When the learning rate is a decreasing throughout training, we show that Algorithm 1 also converges. Given the Lipschitz-continuous gradient and the Bounded variance assumption and a decreasing learning rate sequence η_t satisfying $\eta_t = \eta/1 + t, \forall t \in \{0, 1, \ldots, T-1\}$ and $\eta L \le 1$, if we assume the optimal solution that minimizes the error function $f(w)$ is w^* and let $H_T = \sum_{t=0}^{T-1}(\eta_t - L\eta_t^2/2)$, then the output of Algorithm 1 satisfies the following inequality:

$$\frac{1}{H_T}\sum_{t=0}^{T-1}\left(\eta_t - \frac{L\eta_t^2}{2}\right)\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 \le \frac{f\left(w^0\right) - f\left(w^*\right)}{H_T} + \frac{\sum_{t=0}^{T-1}\eta_t^2 LKM}{H_T}. \tag{C.10}$$

The above property can be proved as follows. When $\eta_t = \eta/1 + t, \forall t \in \{0, 1, \ldots, T-1\}$ and $\eta L \le 1$, taking expectation inequality (C.1) and summing it from $t = 0$ to $T-1$, we have

$$\mathbb{E}\left[f\left(w^T\right)\right] - f\left(w^0\right) \le -\sum_{t=0}^{T-1}\left(\eta_t - \frac{L\eta_t^2}{2}\right)\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 + \sum_{t=0}^{T-1}L\eta_t^2 KM. \tag{C.11}$$

Suppose that w^* is the optimal solution for $f(w)$; then, $f(w^*) - f(w^0) \le \mathbb{E}[f(w^T)] - f(w^0)$. Let $H_T = \sum_{t=0}^{T-1}(\eta_t - L\eta_t^2/2)$, and we have

$$\frac{1}{H_T}\sum_{t=0}^{T-1}\left(\eta_t - \frac{L\eta_t^2}{2}\right)\mathbb{E}\left\|\nabla f\left(w^t\right)\right\|_2^2 \leq \frac{f\left(w^0\right) - f\left(w^*\right)}{H_T} + \frac{\sum_{t=0}^{T-1}\eta_t^2 LKM}{H_T}. \tag{C.12}$$

This proves inequality (C.10).

Since $\eta_t = \eta_0/1 + t$, the learning rate requirements in [43] are satisfied that $\lim_{T\longrightarrow\infty}\sum_{t=0}^{T-1}\eta_t = \infty$ and $\lim_{T\longrightarrow\infty}\sum_{t=0}^{T-1}\eta_t^2 < \infty$. Therefore, when $T \longrightarrow \infty$, the right-hand side of inequality (C.10) converges to 0.

Suppose w^s is chosen randomly from $\{w^t\}_{t=0}^{T-1}$ with probability proportional to $\{\eta_t\}_{t=0}^{T-1}$. According to inequality (C.10), we can prove that Algorithm 1 guarantees convergence to critical points for the nonconvex problem:

$$\lim_{s\longrightarrow\infty}\mathbb{E}\left\|\nabla f\left(w^s\right)\right\|_2^2 = 0. \tag{C.13}$$

D. Memory Usage Comparison

The pipelining scheme in this work uses less memory compared to PipeDream, although it introduces more memory overhead compared to GPipe. PipeDream saves intermediate activations during training, and so does our scheme. However, PipeDream also saves multiple copies of a network's weights for weight stashing, increasing the memory footprint further.

The memory footprint increase due to weight stashing depends on the total weight memory compared to activation memory, the number of active minibatches in the training pipeline, the minibatch size, and the training dataset.

When the weight memory is smaller than the activation memory for a given minibatch size, the memory increase due to weight stashing is not significant. For example, PipeDream's memory increase percentage is only 1% worse than ours for ResNet-20 even though 4 copies of weights would be saved by PipeDream, as shown in Table 9 (*torchsummary* in *PyTorch* is also used to report memory usage for weights and activations for a network and to calculate the additional memory required by the additional copies of activations and weights). This result also holds for ResNet with other depths since the amount of weights and activations grows linearly with the depth of the network.

However, when the weight memory is larger than activation memory for a given minibatch size, weight stashing will have a significant impact on memory footprint. For AlexNet and VGG-16 trained on CIFAR-10, in which the weight memory is much larger than the activation memory, with a minibatch size of 128 using a 4-stage pipelined training, additional 4 copies of weights must be saved due to weight stashing, one per active minibatch in the pipeline, resulting in much more memory increase: a 214% increase in memory footprint that is 177% more than ours (37%) for AlexNet and a 124% increase in memory footprint that is 49% more than ours (75%), as shown in Table 9.

The minibatch size also has an impact on the memory footprint because it directly influences the total amount of activation memory required during training: the larger the minibatch size, the more the activation memory required. Figure 8 shows the memory increase percentage for our scheme and that of PipeDream as a function of minibatch size for the 4-stage pipelined training of LeNet-5, AlexNet, VGG-16, and ResNet-20 in Table 9. When the minibatch size is small, weight stashing has a significant impact for all networks on memory. As the minibatch size increases, the memory increase for ours and PipeDream is similar for ResNet-20. However, for AlexNet and VGG-16, PipeDream still requires more memory then ours due to weight stashing.

Moreover, the input size affects the memory footprint due to weight stashing because it directly affects the amount of activation and weight memory: the larger the input size, the more the activation and weight memory required. Figure 9 shows the memory increase percentage for our scheme and PipeDream as a function of batch size for the 4-stage pipelined training of VGG-16 on ImageNet [44]. For a minibatch size of 32, PipeDream uses 28% more memory than ours due to weight stashing (PipeDream uses a minibatch size of 32 for the training of VGG-16 on ImageNet).

Data Availability

The work uses publicly available models, and the obtained performance data can be made available upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by grants from Natural Sciences and Engineering Research Council of Canada (NSERC) and Huawei Research.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, June 2016.

[2] S. Wang, Y. Guo, Y. Wang, H. Sun, and J. Huang, "Smiles-bert: large scale unsupervised pre-training for molecular property prediction," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pp. 429–436, ACM, Niagara Falls, NY, USA, September 2019.

[3] D. Arya and M. Worring, "Exploiting relational information in social networks using geometric deep learning on hypergraphs," in *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval, ICMR'18*, pp. 117–125, Yokohama, Japan, June 2018.

[4] S. Ben Atitallah, M. Driss, W. Boulila, and H. Ben Ghézala, "Randomly initialized convolutional neural network for the recognition of Covid-19 using X-ray images," 2021, https://arxiv.org/abs/2105.08199.

[5] W. Boulila, M. Sellami, M. Driss, M. Al-Sarem, M. Safaei, and F. A. Ghaleb, "RS-DCNN: a novel distributed convolutional-neural-networks based-approach for big remote-sensing image classification," *Computers and Electronics in Agriculture*, vol. 182, Article ID 106014, 2021.

[6] W. Boulila, H. Ghandorh, M. A. Khan, F. Ahmed, and J. Ahmad, "A novel CNN-LSTM-based approach to predict

urban expansion," *Ecological Informatics*, vol. 64, Article ID 101325, 2021.

[7] X. Chen, E. Adam, G. Li, D. H. Yu, and S. Frank, "Pipelined backpropagation for context-dependent deep neural networks," in *Proceedings of the Interspeech ISCA's 13th Annual Conference 2012*, Portland, OR, USA, September 2012.

[8] Y. Huang, Y. Cheng, A. Bapna et al., "Gpipe: efficient training of giant neural networks using pipeline parallelism," 2019, https://arxiv.org/abs/1811.06965.

[9] D. Narayanan, A. Harlap, A. Phanishayee et al., "Pipedream: generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Huntsville, Canada, October 2019.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[11] H.-Y. Cheng, C.-C. Chen, and C.-L. Yang, "Efficient and robust parallel DNN training through model parallelism on multi-GPU platform," 2019, https://arxiv.org/abs/1809.02839.

[12] Z. Huo, B. Gu, Q. Yang, and H. Huang, "Decoupled parallel backpropagation with convergence guarantee," in *Proceedings of the 35th International Conference on Machine Learning, ICML'2018*, vol. 80, pp. 2103–2111, Stockholm, Sweden, 2018.

[13] H. Mostafa, P. Bruno, Sadique Sheik, and G. Cauwenberghs, "Hardware-efficient on-line learning through pipelined truncated-error backpropagation in binary-state networks," *Frontiers in Neuroscience*, vol. 496, 2017.

[14] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting distributed synchronous SGD," in *Proceedings of the International Conference on Learning Representations Workshop Track 2016*, San Juan, Puerto Rico, May 2016.

[15] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. X.. GeePS, "Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proceedings of the EuroSys'16: Proceedings of the Eleventh European Conference on Computer Systems*, vol. 4, no. 1–4, p. 16, London, UK, April 2016.

[16] J. Dean, G. Corrado, R. Monga et al., "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, vol. 25, Lake Tahoe, NV, USA, December 2012.

[17] P. Goyal, P. Dollár, G. Ross et al., "Accurate, large Minibatch SGD: training ImageNet in 1 hour," 2017, http://arxiv.org/abs/1706.02677.

[18] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and Ion Stoica, "Blink: fast and generic collectives for distributed ml," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 172–186, 2020.

[19] H. Zhang, Z. Zheng, S. Ho, and W. Poseidon, "An efficient communication architecture for distributed deep learning on GPU clusters," in *Proceedings of the USENIX ATC'17: Conference on Usenix Annual Technical Conference (ATC)*, pp. 181–193, Santa Clara, CA, USA, July 2017.

[20] T. Chilimbi, Y. Suzue, A. Johnson, and K. Kalyanaraman, "Project adam: building an efficient and scalable deep learning training system," in *OSDI'14: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pp. 571–582, Broomfield, CO, USA, October 2014.

[21] K. K. Jin, Q. Ho, S. Lee et al., "A distributed framework for scheduled model parallel machine learning," in *Proceedings of the Eleventh European Conference on Computer Systems*, London, UK, April 2016.

[22] S. Lee, K. Jin, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," *Advances in Neural Information Processing Systems*, vol. 27, pp. 2834–2842, 2014.

[23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[24] A. Petrowski, G. Dreyfus, and C. Girault, "Performance analysis of a pipelined backpropagation parallel algorithm," *IEEE Transactions on Neural Networks*, vol. 4, no. 6, pp. 970–981, 1993.

[25] Z. Huo, B. Gu, and H. Huang, "Training neural networks using features replay," in *Proceedings of International Conference on Neural Information Processing Systems*, Montreal, Canada, December 2018.

[26] L. Guan, W. Yin, D. Li, and X. Lu, "Xpipe: efficient pipeline model parallelism for multi-GPU DNN training," 2020, https://arxiv.org/abs/1911.04610.

[27] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster, "Pipelined backpropagation at scale: training large models without batches," 2021, https://arxiv.org/abs/2003.11666.

[28] J. H. Park, G. Yun, C. M. Yi et al., "HetPipe: enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 307–321, Philadelphia, PA, USA, July 2020.

[29] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[30] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and S. Alexander, "Pipe-SGD: a decentralized pipelined SGD framework for distributed deep net training," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Vol. 31, Curran Associates, Inc., Red Hook, NY, USA, 2018.

[31] Y. Jia, E. Shelhamer, J. Donahue et al., "Caffe: convolutional architecture for fast feature embedding," in *Proceedings of the International Conference on Multimedia ICM'2014*, pp. 675–678, Berkeley, CA, USA, June 2014.

[32] P. Adam, S. Gross, S. Chintala et al., "Automatic differentiation in PyTorch," in *Proceeding of the Neural Information Processing Systems Autodiff Workshop*, Long Beach, CA, USA, 2017.

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

[34] Y. Le Cun and C. Cortes, "MNIST handwritten digit database," 1996, http://yann.lecun.com/exdb/mnist/.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 1097–1105, 2017.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*, San Diego, CA, USA, May 2015.

[37] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (Canadian institute for advanced research)," 2018, http://www.cs.toronto.edu/%20kriz/cifar.html.

[38] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-100 (Canadian institute for advanced research)," 2018, http://www.cs.toronto.edu/%20kriz/cifar.html.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language

understanding," in *Proceedings of the of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4171–4186, Minneapolis, MN, USA, June 2019.

[40] M. Naumov, D. Mudigere, H.-J. Michael Shi et al., "Deep learning recommendation model for personalization and recommendation systems CoRR, abs/1906," 2019, http://arxiv.org/abs/1906.00091.

[41] Xilinx, "The xilinx machine learning (Ml) suite, xdnn," 2019, https://github.com/Xilinx/ml-suite.

[42] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

[43] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: a large-scale hierarchical image database," in *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Miami, FL, USA, June 2009.