Parallelizing DNN Training on GPUs: Challenges and Opportunities

Weizheng Xu
University of Pittsburgh
Pittsburgh, PA, USA
wex43@pitt.edu

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Xulong Tang
University of Pittsburgh
Pittsburgh, PA, USA
tax6@pitt.edu

ABSTRACT

In recent years, Deep Neural Networks (DNNs) have emerged as a widely adopted approach in many application domains. Training DNN models is also becoming a significant fraction of the datacenter workload. Recent evidence has demonstrated that modern DNNs are becoming more complex and the size of DNN parameters (i.e., weights) is also increasing. In addition, a large amount of input data is required to train the DNN models to reach target accuracy. As a result, the training performance becomes one of the major challenges that limit DNN adoption in real-world applications. Recent works have explored different parallelism strategies (i.e., data parallelism and model parallelism) and used multi-GPUs in datacenters to accelerate the training process. However, naively adopting data parallelism and model parallelism across multiple GPUs can lead to sub-optimal executions. The major reasons are i) the large amount of data movement that prevents the system from feeding the GPUs with the required data in a timely manner (for data parallelism); and ii) low GPU utilization caused by data dependency between layers that placed on different devices (for model parallelism).

In this paper, we identify the main challenges in adopting data parallelism and model parallelism on multi-GPU platforms. Then, we conduct a survey including recent research works targeting these challenges. We also provide an overview of our work-in-progress project on optimizing DNN training on GPUs. Our results demonstrate that simple-yet-effective system optimizations can further improve the training scalability compared to prior works.

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; • **Computer systems organization → Multiple instruction, multiple data**.

KEYWORDS

deep neural networks, data parallelism, model parallelism, GPUs

ACM Reference Format:
Weizheng Xu, Youtao Zhang, and Xulong Tang. 2021. Parallelizing DNN Training on GPUs: Challenges and Opportunities. In *Companion Proceedings of the Web Conference 2021 (WWW '21 Companion), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3442442.3452055

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW '21 Companion, April 19–23, 2021, Ljubljana, Slovenia
© 2021 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-8313-4/21/04.
https://doi.org/10.1145/3442442.3452055

1 INTRODUCTION

With the capability to provide unprecedented accuracy, Deep Neural Networks (DNNs) have emerged as one of the most popular techniques to solve real-world problems. However, training DNN models is very computing resource-demanding and time-consuming [8, 9, 46]. For example, training a large BERT model takes up to 3 days on 16 Google TPUs [13] and it takes more than 40 days to train an AlphaGo Zero system [49]. As a result, training large-scale DNN models is generally deployed in datacenters and occupies thousands of machine-hours. As reported by recent study, training DNN is becoming a significant fraction of the datacenter workload, and optimizing/accelerating the training process has significant cost, resource and SLA implications [18, 31].

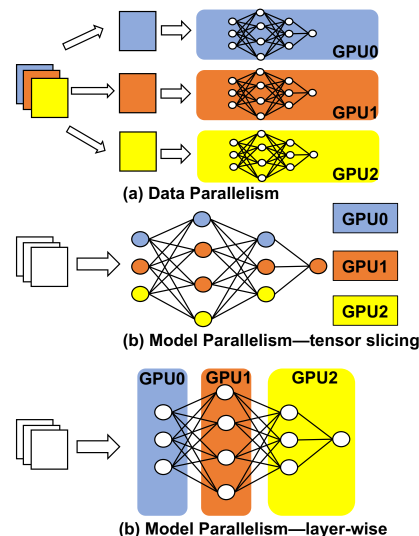

Figure 1: Data parallelism and model parallelism on multi-GPU systems.

To accelerate the DNN training process, multi-GPU in datacenters are adopted with two types of parallelization strategies (*data parallelism* and *model parallelism*) [25, 33]. In data parallelism (shown in Figure 1(a)), the input data is split into mini-batches and the mini-batches are distributed across GPUs. DNN model weights are replicated on each GPU and synchronization is performed at the end of each batch processing to gather the gradients and update the model weights [26]. For model parallelism, there are two incarnations: tensor slicing and layer-wise [40, 44]. As shown in

Figures 1(b) and (c), tensor slicing horizontally partitions the model by neurons in each layer, whereas layer-wise splits the DNN model into different subsets of neural network layers. Different model partitions are computed on separate GPUs and the intermediate computation results at the layer boundaries are transferred between GPUs. In practice, model parallelism is generally combined with pipelining the computations from consecutive batches to achieve high training throughput on multiple GPUs [14, 15, 21, 47]. We refer to this method as *pipeline parallelism* [3, 23, 45, 47]. According to how the weights are updated, existing pipeline parallelism approaches can be classified into two categories: synchronous pipeline parallelism [20] and asynchronous pipeline parallelism [29]. We will further investigate them in section 2 and section 3.

While data parallelism and pipeline parallelism speed up the training process, naively adopting them across multiple GPUs can lead to *non-scalable* execution times. We observe that there is no one-size-fits-all parallelization strategy due to the DNN intrinsic challenges. These challenges limit the scalable training performance that one can obtain by using multiple GPUs in datacenters. In this paper, we first provide a detailed summary of the challenges and include a survey of recent efforts in addressing the challenges. Then, we discuss our on-going project with several simple-yet-effective approaches to improve the training performance.

2 CHALLENGES

Challenge 1: Data movement overhead. The data parallelism involves different types of data movements among GPUs [11, 43]. Specifically, each GPU has to wait for the mini-batch data to be transferred and model weights to be broadcasted before starting forward propagation. Also, after backward propagation, the gradients on each GPU need to be gathered and then used to update the model weights. Therefore, adopting data parallelism across multiple GPUs involves expensive data movement, leading to GPU under-utilization and performance degradation. For pipeline parallelism, although there exists data movement between GPUs at the layer boundaries, it can be effectively hidden by using the asynchronous memory copy that overlaps the data transfer with computations [34]. Therefore, data movement overhead is less dominating in pipeline parallelism compared to it in data parallelism.

Challenge 2: Load imbalance and straggler effect. Pipeline parallelism allows training large models whose model size exceeds single GPU memory capacity. It achieves that by partitioning the model among different GPUs. However, each partition may have different amount of computation and communication loads, leading to imbalanced workloads across different GPUs [29]. Such imbalance may cause under-utilization of GPU resources. More importantly, when the resource configurations (e.g., computation power and memory space) of the GPUs are heterogeneous or dynamic (always happen in the datacenter environment), the straggler effect is another non-negligible reason that causes poor training performance [34]. Note that, load imbalance is not a problem in data parallelism, as the workload of each GPU is approximately equal.

Challenge 3: Synchronization overhead. As we mentioned in section 1, existing pipeline parallelism approaches can be roughly classified into synchronous pipeline parallelism and asynchronous pipeline parallelism, based on the way that the weights are updated. The synchronous pipeline parallelism requires necessary gradient synchronizations between adjacent training iterations[1] to ensure convergence without accuracy degradation. Because each GPU only needs to maintain the weights update of its own partition, the synchronizations are implemented within each GPU. Recall our discussion in pipeline parallelism where intermediate results are transferred across GPUs, therefore, there are many "bubble areas" (GPU idleness) in the execution pipeline due to the intermediate data transfer and the within GPU synchronizations. For data parallelism, most of the synchronization overheads are caused by data movements and we have considered them in data movement overhead.

Challenge 4: Weight inconsistency and weight staleness. Asynchronous pipeline parallelism updates the weights asynchronously during training. In general, asynchronous pipeline parallelism improves pipeline utilization and has a better performance compared to the synchronous approach. However, it incurs weight inconsistency or weight staleness issues due to the cross-training of multiple batches. Training with weight inconsistency means that the forward pass and backward pass of one batch use different versions of weights. For weight staleness, let us assume that a model is split into n partitions. Under weight staleness, the gradients in the 1st weight partition are computed using the version of the 1st partition that are updated n iterations before (i.e., the "stale" weights). Similarly, gradients in the 2nd partition are computed with weights that are $n-1$ iterations before. This discrepancy in weight versions can slow down the training convergence. As a result, weight staleness and weight inconsistency lead to training instability and accuracy loss.

Challenge 5: Insufficient GPU memory. As GPUs generally feature a small capacity of high-bandwidth memory compared to CPU host memory, both data parallelism and pipeline parallelism may suffer from insufficient GPU memory when training DNNs with large model size and/or large input data size. In data parallelism, DNNs with large model sizes cannot be deployed on the GPUs since models have to be duplicated across the GPUs. In synchronous pipeline parallelism, it tries to schedule as many batches as possible to the first GPU at once. However, this may significantly increase the memory requirement for holding intermediate computation results of all concurrent batches. In asynchronous pipeline parallelism, it needs to keep multiple versions of weights on each GPU to avoid weight inconsistency [29]. Therefore, more memory is required to store different weight versions.

3 A SURVEY OF PARALLELIZING DNN TRAINING

3.1 Reducing Data Movement

Data movement is one of the main challenges in data parallelism. Several prior works focus on reducing data movement overhead [6, 10, 12, 16, 17, 19, 32, 38, 42]. Among these efforts, model compression has been demonstrated as an effective method to reduce the memory requirement for large DNN models. Specifically, the parameter pruning [16, 17] and quantization [10] based methods explore the redundancy in the model parameters and remove

[1] An iteration is defined as processing one batching including forward pass and backward pass.

Table 1: List of recent works.

Challenges	Related works
Data movement overhead	[6, 10, 12, 16, 17, 19, 32, 38, 42, 43]
Load imbalance and straggler effect	[14, 20, 28–30, 41, 50]
Synchronization overhead	[2, 20, 22, 35, 36]
Weight inconsistency and weight staleness	[5, 15, 24, 27, 29, 34, 45, 48]
Insufficient GPU memory	[4, 7, 20, 21, 29, 30, 37, 39]

the redundant parameters that are not sensitive to the performance. For example, Deep Compression [17] quantizes the weights and uses Huffman coding to encode the quantized weights. CLIP-Q [42] proposes joint network pruning and weight quantization in a single learning framework. However, their approaches require more iterations to converge. Low-rank factorization [12] uses matrix/tensor decomposition to estimate the informative parameters of the DNNs. The knowledge distillation [19] method learns a distilled model and trains a compact neural network to reproduce the output of a larger network. These approaches achieve significant data movement reduction. However, they either require modifications of the network architectures or ineffective in training on multi-GPU platforms.

3.2 Optimizing Load Imbalance and Straggler

In pipeline parallelism, the DNN model is partitioned into a group of layers and placed on different GPUs. Different partitioning algorithms can significantly affect the overall training performance. At a high-level, two types of optimizations have been explored by prior works.

Static partitioning. An intuitive method for workload balancing is to partition the DNN model into groups of layers in a topology-aware manner such that each group completes at a similar rate. Static partitioning is conducted based on the DNN network topology and the hardware configuration before training. It has been used in some recent works [29, 30, 50]. However, there are two potential problems in static partitioning: 1) it is difficult to evenly partition the model due to the size variance of parameters in different layers; and 2) when the execution is dynamic or and the platform is heterogeneous, the straggler effect diminishes the training efficiency.

Dynamic partitioning. Dynamic partitioning is more favored in training in the cloud where the cloud resources exhibit heterogeneity and diversity. Elaticpipe [14] proposes an auto-tuning and flexible partitioning mechanism that can periodically collect the performance information and redistribute the workload for each device during runtime. As a result, their approach is able to "auto" adapt to the dynamic and heterogeneous cloud environment. This method explores the FLOP-based workload modeling, but extra scheduling mechanisms are needed.

3.3 Reducing Synchronization Overhead

The synchronous pipeline parallelism requires necessary gradient synchronizations between adjacent training iterations. The All-Reduce [35] approach reduces the synchronization overhead, but with a cost of extra data movement. There are several scheduling approaches [2, 22, 35] that try to delay the synchronizations. For example, P3 [22] synchronizes the parameter slices based on their priority, where the priority of a slice is determined by the time it is required again in the subsequent iteration. Such an approach can utilize the available resources more efficiently. However, it needs modifications to the communication mechanisms and maintains a priority queue.

3.4 Mitigating Weight Inconsistency and Staleness

The accuracy in pipeline parallelism suffers from weight staleness and inconsistency issues [29, 48]. PipeDream [29, 30] resolves weight inconsistency by leveraging weight stashing that keeps multiple versions of weights. However, it suffers from weight staleness issue and requires more epochs in training to reach convergence. Additionally, more memory is required in weight stashing, leading to failure of training larger DNN using PipeDream. Another approach is weight prediction which addresses the weight inconsistency and weight staleness issues in the asynchronous pipeline parallelism [5, 15]. For example, SpecTrain [5] uses the same pipeline structure as PipeDream. Instead of storing the weights for each mini-batch in the pipeline, it uses the smoothed gradients multiplying the differences between weight versions to predict the future weights. This method is based on the observation that the smoothed gradients used in Momentum Stochastic Gradient Descent (SGD) can reflect the trend of weight updates. Compared with SpecTrain, XPipe [15] uses Adam-based weight prediction instead of Momentum SGD based approach to provide a more effective solution. Nevertheless, there is no single weight prediction method that can have comparable performance as synchronous training. In Table 1, we also summarize other works that focus on weight inconsistency and weight staleness.

3.5 Addressing Memory Insufficiency

Existing data parallelism and pipeline parallelism suffer from limited GPU memory capacity when training with large model size and/or large input data size. For example, the VGG19 with batch size 512 cannot be trained with 4 GTX 1080Ti GPUs due to out of memory issue. To mitigate the memory insufficiency, re-computation is proposed by [7]. It reduces memory usage of intermediate results during DNN training by recomputing the activation instead of storing them in memory. However, such an approach involves extra computation overheads, and the performance relies on a good trade-off between memory usage and computation overhead. GPipe [20] is a representative work that benefits from using re-computation method in pipeline parallelism. vDNN [39] proposes swapping the tensors between CPU and GPU to reduce peak memory consumption. There are also other optimizations from the architectural aspect. GEMS[21] proposes training a replica of the same DNN in an inverted manner based on the observation that the GPU has different memory consumption during the forward and backward passes. As a complementary work, PipeDream-2BW [30] proposes a doublebuffered weight updates (2BW) design to reduce the memory footprint during training. However, these approaches have limited flexibility depending on the GPU platforms they use.

Takeaway. Considering the current works related to the five challenges, we can find that 1) most of the works are "point-solutions" and focus on "single-sourced" performance overheads, and 2) they have limited performance improvements due to the involved extra overheads. Therefore, in this paper, we explore systematic approaches to addressing the challenges with minimized

Table 2: The percentages of execution time spent on data movement (256 batch size). (I) denotes input batch data and (M) denotes model parameters. × represents out-of-memory errors.

Model	1 GPU		2 GPU		3 GPU		4 GPU	
	I	M	I	M	I	M	I	M
ResNet34-256	12.63%	0	22.17%	4.11%	26.20%	5.29%	31.14%	8.57%
ResNet50-256	×	×	×	×	16.89%	3.61%	22.53%	6.52%
VGG19-256	×	×	×	×	8.54%	12.58%	11.29%	21.17%
MobileNet_v2-256	×	×	23.13%	0.64%	30.59%	0.85%	38.17%	1.01%

overheads. We also reveal future optimization opportunities to further improve the scalable training of DNN models on multi-GPU datacenters.

4 OUR APPROACHES AND FUTURE OPPORTUNITIES

Motivated by the aforementioned challenges, we introduce our on-going work targeting to deliver scalable training on multi-GPU platforms. At a high-level, our work focuses on two aspects: 1) data movement in data parallelism and 2) time-to-target accuracy in pipeline parallelism.

As we mentioned earlier, data movement is one of the major challenges in data parallelism that hurt the training performance. To quantify the problem, we characterized the overheads of data movement during training. We classify two types of data movements: i) input batch data and ii) model parameters. Table 2 summarizes the percentages of execution time spent on data movement using four networks under the configurations from 1 GPU to 4 GPUs. One can observe that i) the data movement overheads occupy a large portion of the execution time and ii) the overheads increase with the number of GPUs employed. As a result, the delivered performance hardly scales with the number of GPUs. For example, ResNet34 with a batch size of 256 only achieves 2.2× execution time reduction in four GPUs compared to one GPU. To reduce the data movement overhead, we propose three optimizations: *CPU-centric input batch splitting*, *mini-batch pre-loading*, and *model weights compression*. The first two optimizations employ CPU to dispatch the mini-batches and remove mini-batch transfer from the execution's critical path, enabling overlapping of data transfer and GPU computation. The third optimization explores the similarity among model weights and leverages compression to significantly reduce the data transfer with negligible impact on model accuracy. While we have effectively reduced the data movement overheads, our proposed compression requires extra computations to compress and decompress the weights. We are currently working on reducing the compression overheads and leveraging modern GPU features (e.g., mix-precision training) to efficiently compress and decompress the weights without affecting the accuracy. Moreover, our current project focus on a single-node multi-GPU system. We are planning to extend our project to consider the data movement involved in distributed training in datacenters [1, 27].

As we discussed in section 2 and section 3, load imbalance and weight staleness are two main challenges in asynchronous pipeline parallelism. In our work, we conduct simulation experiments to explore the impact of weight staleness on the time-to-target accuracy when the number of GPUs increases. The experiments are simulated from 1GPU to 16 GPUs. We observed that, as the number of GPU increases, more epochs would be required to reach a target accuracy, especially for 8 GPUs and more. For example, the training with 16 GPUs needs 52 epochs to reach 40% Top-1 accuracy, while the training with 1 GPU only needs 30 epochs. As a result, even though each epoch's execution time is improved by using asynchronous pipeline parallelism, the number of epochs to reach target accuracy is actually increased. That is, the efficiency of each epoch training is reduced. Moreover, multiple versions of weights need to be stored in each GPU to avoid weight inconsistency and staleness, leading to extra memory cost that affects the efficiency of training large-scale model.

Inspiring from our analysis, we highlight that when using a large number of GPUs to train the models (e.g., 8 GPUs and more), the weight staleness and memory usage for storing multiple versions of weights issues are important to address. From the structure view, we will explore to trade memory (with minimized memory overhead) for efficiency to convergence. That is, we use extra memory to store more partitions in each device (e.g., two partitions in each GPU when 4 GPUs are used). With more partitions in each GPU, we can train our model from bi-direction (one training path is from the first GPU to the last GPU and the other path is from the last GPU to the first one). By leveraging this design, we can obtain the latest version of weight to train the new batch of data, thus improving each epoch training's efficiency. Moreover, load imbalance is also one important overhead in pipeline parallelism. We will explore periodically shuffling the partitions among GPUs to balance their workloads. In our current work, we mostly use the layer-wise model parallelism due to its simplicity. However, it is necessary to consider tensor slicing for the scenarios that only a single layer cannot fit into a single device. Thus, given the two types of model parallelism, data parallelism, and pipeline parallelism, we are exploring combining them on the multi-GPU system to train large-scale DNN models.

5 CONCLUSION

Training modern DNN models, especially large and giant models, encounters a "scalability wall" due to the massive data movements, frequent synchronization, and severe load imbalance. Such overheads have been exacerbated by system configurations and model complexities, making the scaling optimizations difficult to derive and implement. As a result, the delivered training performance rarely scales with the increase of computing resources (e.g., number of GPUs) in the system.

In this paper, targeting multi-GPU platforms, we systematically summarize the challenges in data parallelism and pipeline parallelism. Moreover, we provide a survey study of recently published works that target addressing these challenges. We also discuss our work-in-progress project and show promising initial results on reducing the data movement and mitigating the load imbalance and weight staleness.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants #2011146 and a startup funding from the University of Pittsburgh.

REFERENCES

[1] Ammar Ahmad Awan, Jereon Bédorf, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. 2019. Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation. In *CCGRID*. IEEE, 498–507.

[2] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. 2020. Preemptive all-reduce scheduling for expediting distributed dnn training. In *INFOCOM*. IEEE, 626–635.

[3] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.

[4] Zhenkun Cai, Kaihao Ma, Xiao Yan, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2020. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *arXiv preprint arXiv:2004.10856* (2020).

[5] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[6] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[8] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[9] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*. 571–582.

[10] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2016. Towards the limit of network quantization. *arXiv preprint arXiv:1612.01543* (2016).

[11] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[12] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando De Freitas. 2013. Predicting parameters in deep learning. *arXiv preprint arXiv:1306.0543* (2013).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[14] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*. 5–9.

[15] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. 2019. XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training. *arXiv preprint arXiv:1911.04610* (2019).

[16] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic network surgery for efficient dnns. *arXiv preprint arXiv:1608.04493* (2016).

[17] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[21] Arpan Jain, Ammar Awan, Asmaa Aljuhani, Jahanzeb Hashmi, Quentin Anthony, Hari Subramoni, Dhabaleswar Panda, Raghu Machiraju, and Anil Parwani. 2020. GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training. In *SC*. IEEE Computer Society, 621–635.

[22] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *arXiv preprint arXiv:1905.03960* (2019).

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[24] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. 2020. Pipelined Backpropagation at Scale: Training Large Models without Batches. *arXiv preprint arXiv:2003.11666* (2020).

[25] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[27] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[28] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2020. Accelerating distributed inference of sparse deep neural networks via mitigating the straggler effect. In *HPEC*. IEEE, 1–7.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*. 1–15.

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2020. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503* (2020).

[31] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. 2020. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518* (2020).

[32] Bogdan Nicolae. 2020. DataStates: Towards Lightweight Data Models for Deep Learning. In *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 117–129.

[33] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing multi-GPU parallelization strategies for deep learning training. *IEEE Micro* 39, 5 (2019).

[34] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large {DNN} Training on (Whimpy) Heterogeneous {GPU} Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *ATC*. 307–321.

[35] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[36] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*. 16–29.

[37] S. Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. In *SC*.

[38] Brandon Reagan, Udit Gupta, Bob Adolf, Michael Mitzenmacher, Alexander Rush, Gu-Yeon Wei, and David Brooks. 2018. Weightless: Lossy weight encoding for deep neural network compression. In *International Conference on Machine Learning*. PMLR, 4324–4333.

[39] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*. IEEE, 1–13.

[40] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084* (2018).

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[42] Frederick Tung and Greg Mori. 2018. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7873–7882.

[43] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *PPoPP*. 41–53.

[44] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys*. 1–17.

[45] An Xu, Zhouyuan Huo, and Heng Huang. 2020. On the acceleration of deep learning model parallelism with staleness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2088–2097.

[46] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. 2015. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1355–1364.

[47] Letian Zhao, Rui Xu, Tianqi Wang, Teng Tian, Xiaotian Wang, Wei Wu, Chio-in Ieong, and Xi Jin. 2020. BaPipe: Exploration of Balanced Pipeline Parallelism for DNN Training. *arXiv preprint arXiv:2012.12544* (2020).

[48] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. 2019. Dynamic stale synchronous parallel distributed training for deep learning. In *ICDCS*. IEEE, 1507–1517.

[49] Shuai Zheng, Haibin Lin, Sheng Zha, and Mu Li. 2020. Accelerated Large Batch Optimization of BERT Pretraining in 54 minutes. arXiv:2006.13484 [cs.LG]

[50] Wentao Zhu, Can Zhao, Wenqi Li, Holger Roth, Ziyue Xu, and Daguang Xu. 2020. LAMP: Large Deep Nets with Automated Model Parallelism for Image Segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 374–384.