Disruptive Research on Distributed Machine Learning Systems

Guanhua Wang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-83
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-83.html

May 12, 2022

Disruptive Research on Distributed Machine Learning Systems

by

Guanhua Wang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph Gonzalez
Professor Alexandre Bayen
Professor Michael Mahoney

Spring 2022

Disruptive Research on Distributed Machine Learning Systems

Abstract

Disruptive Research on Distributed Machine Learning Systems

by

Guanhua Wang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Deep Neural Networks (DNNs) enable computers to excel across many different applications such as image classification, speech recognition and robotic control. To accelerate DNN training and serving, parallel computing is widely adopted. System efficiency is a big issue when scaling out. High communication overheads and limited on-device memory are two major causes for system inefficiency in distributed machine learning.

This dissertation studies possible ways to mitigate communication bottlenecks and achieve better on-device memory utilization in data and model parallelism for distributed machine learning workloads.

On the communication side, our Blink project mitigates communication bottleneck in data parallel training. By packing spanning trees rather than forming rings, Blink achieves higher flexibility in arbitrary networking environments and provides near-optimal network throughput. To eliminate the communication in model parallel training and inference, we go above from system layer to application layer. Our sensAI project decouples a multi-task model into disconnected subnets, where each subnet is responsible for decision making of a single task or a subset of the original task-set.

Towards better utilization of on-device memory, our Wavelet project intentionally adds task launching latency to interleave peak memory usage across different waves of training tasks on the accelerators. By packing multiple training waves on the same accelerator, it improves both computation and on-device memory utilization.

To my parents, Ying Han and Xin Wang.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my gratitude and appreciation to my academic advisor Professor Ion Stoica. This dissertation would not be possible without the guidance and support from him. Ion was the primary reason I started my PhD at UC Berkeley. One big thing I learnt from Ion is to do fundamentally novel research, either finding new problems or proposing new solutions to old problems. These principles guide me through my whole PhD. For my first few years, even though I did not make good progress, Ion still believed in me. For example, he thought Blink was a good project even after it had been rejected for several years across multiple good venues. I want to thank Professor Joseph Gonzalez for insightful feedback and helping me connect with machine learning folks at Berkeley. I also want to thank Professor Dawn Song for offering me the chance to learn how an early-stage startup operates in the blockchain area. I am also thankful to my dissertation committee members Professor Alexandre Bayen, Professor Michael Mahoney, and my prelim committee Professor Scott Shenker. All of them are role models and I took much inspirations from their works in this dissertation.

I was fortunate to work with excellent mentors during my PhD. I would like to especially thank both Amar Phanishayee (MSR) and Shivaram Venkataraman (UW–Madison). Amar was my mentor during my internship at MSR, who leaded me into the research area of machine learning systems. Shivaram was a final-year PhD student in the AMPLab, who also joint our Blink project later on. Amar, Shivaram and I worked on Blink project for almost 3 years. I learnt a lot from both of them, such as how to set proper baby steps, how to write more readable code, how to make good presentations.

I am grateful to Zhuang Liu for helping me understand concepts and theories in computer vision models. I still miss the days when Zhuang, Brandon Hsieh and I sitting together to solve the problems we encountered in the sensAI project. I learnt a lot from Zhuang during our 2 years collaboration. For instance, anytime we got stuck at some point, he was the person to help us verify the theory and debug the code. We then become good friends. And now I am trying to convince Zhuang to move to Seattle for work, so that we can still meet in-person frequently.

I want to thank all my other collaborators and friends. I would like to thank Dequan Wang, Gur-Eyal Sela, Zhewei Yao, Fisher Yu (ETH) and Professor Kannan Ramchandran for helping us revise our previous paper drafts. I really enjoy having interesting discussions with Paras Jain, Fangyu Wu, Hong Zhang (UWaterloo), Siyuan Zhuang on various topics like deep learning, blockchain, robotic control, computer networks. For my prelim exam preparation, Chang Lan and Peter Xiang Gao helped me a lot. I also enjoy long-distance running with Haoran Tang, Renyuan Xu (USC) from Berkeley to Oakland during the weekends. I am particularly thankful to Wei Bai (MSR), Zhe Cao (Facebook), Wenyu Wang (UIUC), Keyu Wang (Google) for useful tips during my job hunting. I would like to also thank Kenan Jiang, Kehan Wang, Yaoqing Yang, Jichan Chung, Balaji Veeramani, Vipul Gupta, Adarsh Karnati, Zihao Fan, Praveen Batra, Hank O'Brien, Yingxin Kang, Sahil Rao, Aleksander Ficek, Xiangjun Li, Jorgen Thelin (MSR), Nikhil Devanur (Amazon) for contributing to

our code repositories. I also want to thank our administrative staff Kattt Atchley, Boban Zakovich, Shane Knapp, Jon Kuroda, Dave Schonenberg in the AMPLab/RISELab. I cannot list all the people who are important to me. But I would like to thank all the folks who spending time with me during my PhD.

Last but not least, I would like to thank my parents Ying Han and Xin Wang for their unconditional support and selfless love throughout my whole life.

Chapter 1

Introduction

1.1 Motivation

Machine learning (ML) becomes one of the paramount technologies in recent decades [1]. Machine learning [2] and Artificial Intelligence (AI) [3] in general are regarded as major components for the fourth Industrial revolution in the human history [4].

To handle complicated tasks in computer vision [5][6][7], speech recognition [8][9] and robotic control [10], deep neural networks (DNNs) stands out and becomes the main force in machine learning area over the last decade. Influential models like AlexNet [5], BERT [8], AlphaGo [11] enable computers to excel over human in a broad spectrum of tasks like image classification, natural language processing and game playing.

To achieve better intelligence and higher model serving accuracy, both the input data size [12] and deep learning model size [13] are growing drastically. Taking computer vision area as an example, from CIFAR-10/100 [14] to ImageNet-1K [15] dataset, the number of images increase from 60k to over 1.2 million. In addition, the image resolution also grows a lot from CIFAR's 32x32 pixels to ImageNet's 256x256 pixels. On the model side, from GoogLeNet [16] to recent AmoebaNet series [17], the number of parameters almost increase 200 times. More astonishing numbers can be found in the Natural Language Processing (NLP) domain [18][19]. Taking transformer-based model GPT-3 [13] as an example, the number of parameters can be up to 175 billions. And the model is trained for around 300 billion tokens.

Given these giant models and enormous amount of input data, it is almost impossible to conduct model training or serving on a single accelerator. Thus, distributed model training and serving paradigms are widely adopted [20][21]. These distributed machine learning systems mainly focus on collectively using multiple accelerators (e.g., GPUs) to execute in-parallel model training and serving tasks [22][23]. System efficiency is a big issue when scaling out to tens, hundreds or even thousands of accelerators. More specifically, high communication overhead and limited on-device memory are two major causes of system inefficiency in large-scale machine learning systems.

In this thesis, we study possible ways to improve system efficiency in both communication and on-device memory aspects. We aim to mitigate network communication bottleneck and improve on-device memory usage for distributed machine learning workloads. We next summarize the goals and contributions of this dissertation.

1.2 Thesis Contributions

For distributed DNN training and serving, two main parallel paradigms are data parallelism and model parallelism. In data parallelism, each machine or accelerator holds a full copy of the model and conduct local training or serving on a dis-joint subset of the input data [24][25]. In model parallelism, each device only maintains one partition of the whole model and executes training or serving on the same shared input data [26][20][27].

Model synchronization is a major communication overhead in data and model parallel training. In data parallelism, since all the workers are trained on separate input data partitions, model synchronization is needed to synchronize model parameters among all the workers involved. In recent model parallel training system like Megatron-LM [28], synchronization using collectives (e.g., All-Reduce) is also required to aggregate partial matrix-multiplication results. Collective communication is the main-stream scheme for model synchronization in modern distributed machine learning frameworks [29][23][30]. Different companies provide their own collective communication libraries, such as NCCL [31] from Nvidia, Horovod [32] from Uber and Gloo [33] from Facebook. All of them focus on building rings in the given network topology to conduct collective communication. Our Blink project [34] adopts a different approach with topology-awareness. By packing maximum number of spanning trees in the topology, we achieve higher link utilization than ring-based schemes. Our Blink solution can also achieve near-optimal network throughput performance given arbitrary network topology and heterogeneity.

In model parallel training and serving, different workers also need to communication intermediate results like activations and gradients [20][35][36] in each iteration. To elimiate the communication among different nodes holding different model partitions, we propose sensAI project [37]. By applying a divide-and-conquer paradigm, we split a multi-task model into a bunch of disconnected subnets, where each is responsible for decision making of a single task.

Lots of DNN training and serving jobs are memory bounded [38][39]. To achieve higher accelerator utilization, previous literature propose to multiplex multiple jobs on each accelerator [38][40]. However, job-multiplexing on the same device introduces extra overheads, such as frequent context switching [41] or data loading from disk storage [42], inter-job interference [38], extra memory footprints for holding multiple models inside device memory [40]. More importantly, these job-multiplexing schemes cannot speed-up the training progress of a single job. Our Wavelet project [43] provides an generic and efficient way to improve accelerator utilization in the single job case. By intentionally adding task launching latency,

we interleave peak memory usage among different training waves of a single job. Thus we can improve both computation and on-device memory usage in the single job case.

In this thesis, we incorporate our previous research work in distributed machine learning systems, namely Blink [34], sensAI [37] and Wavelet [43]. For accelerators, we limit our discussion to Nvidia GPUs [44]. And our approaches should be generally applicable to other hardware accelerators like TPUs [45], FPGAs [46] and other ASIC chips [47][48].

Contributions: We summarize the main contributions of this thesis as follows:

- By packing spanning trees rather than forming rings, Blink achieve higher throughput than ring-based solutions given arbitrary network environments.
- To eliminate the communication in model parallelism, we propose sensAI as an divide-and-conquer approach. By decoupling a multi-task model into multiple disconnected single-task subnets, we remove the communication among these subnets.
- By interleaving peak memory usage among multiple training waves of a single job, Wavelet improves accelerator utilization on both computation side and on-device memory side.

1.3 Thesis Organization

This thesis is organized as follows. Starting with Chapter 1, we discuss the motivation of this dissertation and our main contributions.

Chapter 2 provides more detailed explanations of deep learning models, distributed paradigms. We first describe training and serving stages of deep learning models. Then we discuss two main distributed methods as data parallelism and model parallelism.

Chapter 3 studies collective communication schemes, which is the modern model synchronization method for both data and model parallel training. The key issue of existing ring-based solution is some hardware links would be wasted if these links cannot form a new ring. We develop Blink [34] to tackle this issue. We abandon ring-based solution and use spanning trees to achieve higher flexibility and provide near-optimal performance given arbitrary network environments.

Following that Chapter 4 presents a new technique to decouple DNN model into disconnect subnets. Thus, we can eliminate communication in model parallel training and inference. Here we mainly explain our sensAI [37] approach in Convolution Neural Networks (CNNs) setting.

We next study how to improve GPU memory utilization in single DNN training job case as Chapter 5. We argue that gang scheduling policy [49][50][51] may under-utilize the limited on-device memory. In Wavelet project [43], we propose a novel yet simple scheduling policy called tick-tock. We profile and interleave peak memory usage among multiple training waves on the same group of GPUs. Thus, it can achieve near-optimal device memory usage and consequently increase the computation core usage.

Finally, Chapter 6 discusses future research directions for distributed machine learning systems. We then conclude this thesis with a summary of our main results.

Chapter 2

Background

2.1 Deep Learning Models

Deep learning becomes the hottest topic in machine learning research for the last decade. Deep learning models in computer vision, natural language processing and reinforcement learning enable machine to excel across a wide range of difficult tasks such as object detection [52][53], question answering [8][54] and robotic control [10][55].

As shown in Figure 2.1, A normal DNN model consists of input layer, several hidden layers and an output layer. And each layer includes several neurons, where each neuron can often be regarded as a weight matrix. Communication is usually needed between adjacent layers of the model. And normally no communication is needed among neurons within each layer.

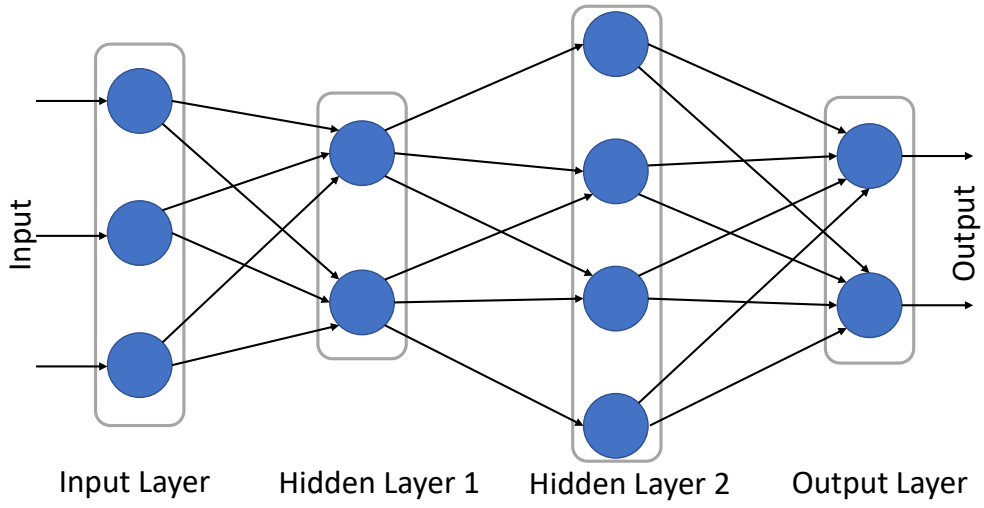


Figure 2.1: A toy example of normal deep learning model.

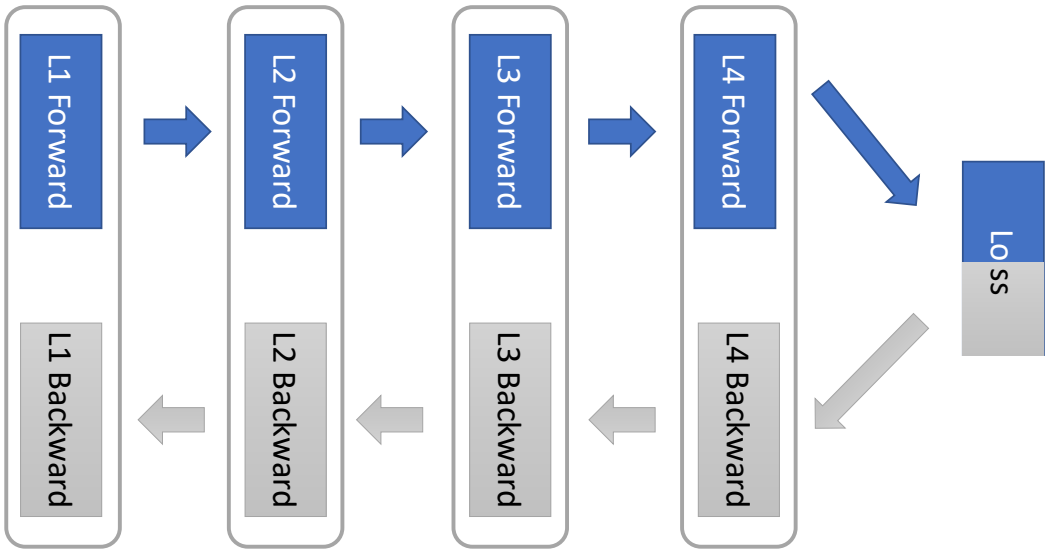


Figure 2.2: Forward propagation and backward propagation in DNN model training.

For instance, in CNN models, the model layers are mainly convolution layers [56] with Max-Pooling and ReLU (Rectified Linear Unit) layers in between [57]. And the models usually end with several fully-connected (FC) layers. To solve the internal covariate shift issue, modern ConvNets adopt Batch Normalization (BN) [58] to improve model serving accuracy. In deep Reinforcement Learning (RL) domain, deep RL models are mainly composed of fully-connected layers [10], but can also incorporate other layers like convolution layers [59], recurrent layers [60] and transformer layers [61][62].

Besides data pre-processing like tokenization [63] or image padding [57], deep learning models usually have two main stages, namely training and serving. We next explain each of them in Section 2.1.1 and Section 2.1.2 respectively.

2.1.1 Training

Given a specific model structure, the model parameters are initialized with random weights values. Then the training iterations start by consuming batches of input data and pass it into the model.

For a single batch of input, the training iteration mainly contains two steps: forward propagation and backward propagation [64].

As depicted in Figure 2.2, forward propagation [64] follows the blue arrows on the upper half of the figure. And backward propagation is shown as the grey arrows of the bottom half in in Figure 2.2. Here L1, L2, L3, L4 refer to the DNN model layers from 1 to 4.

In forward propagation shown in Figure 2.2, given one training input batch, the data first go through L1. After L1 generates local activations, it passes its layer outputs to L2 and then triggers L2's forward propagation, and so on and so forth. After L4 finishes forward propagation, its output values $\hat{y}$ go through the loss function module.

The goal of model training is to find proper weights and biases θ given a loss function on the training data. If the model serving task is classification, cross entropy loss or negative log-likelihood is often used. Our training target is to minimize cross entropy loss function as:

$$\mathcal{L}(\theta) = -\sum_{i}^{n} y_i log(\hat{y}_i) \tag{2.1}$$

As illustrated in Equation 2.1, y_i are the true labels whereas $\hat{y}_i$ are the prediction probabilities.

For regression problems, mean square error (MSE) loss is commonly used. And our target is trying to minimize MSE loss function as:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{2.2}$$

Similar as Equation 2.1, in Equation 2.2, y_i are truly observed values whereas $\hat{y}_i$ are predicted values.

After loss calculation, we can conduct backward propagation [64] in the reverse direction (i.e., grey arrows in Figure 2.2). The backward propagation starts from L4. After L4 generates its local gradients, it pass its gradient outputs back to L3, and so on and so forth. Then we can use gradients generated in each layer to update model weights in order to minimize our loss function.

For each batch of input, a training iteration is completed after one forward pass followed by one backward pass.

2.1.2 Serving

Compared with model training in Section 2.1.1, model serving stage is much simpler. Basically it only conducts the forward propagation shown in Figure 2.2 and generates prediction values for model serving.

Latency is the key performance indicator at the model serving stage [65][66][67][68]. Big tech companies try to reduce model serving latency by providing specific frameworks. For example, Google proposes TensorFlow Runtime [69] for low-latency model inference. Facebook/Meta's PyTorch incorporates ONNX runtime [70] to reduce model serving latency. Nvidia also provides TensorRT [71] SDK for accelerating model serving on the GPU hardware.

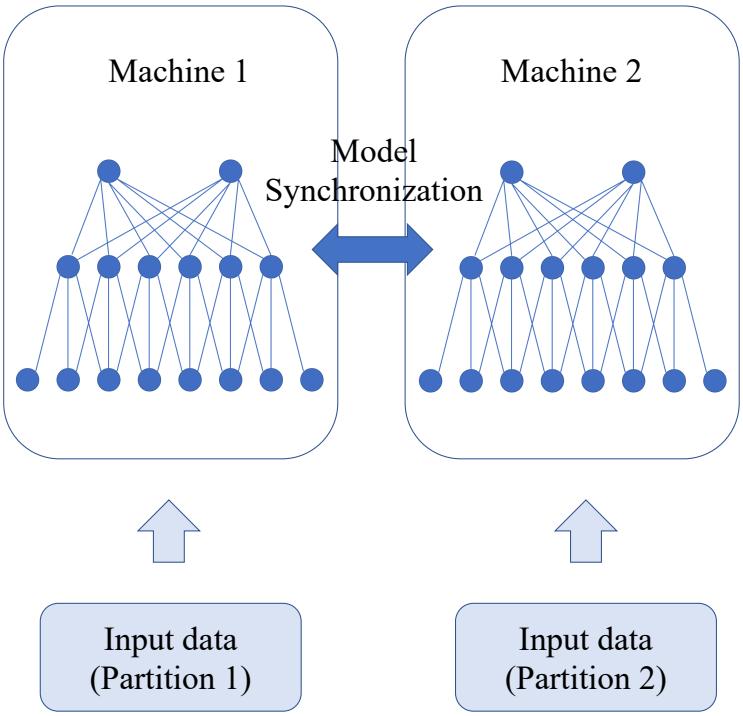


Figure 2.3: Data parallel training paradigm.

2.2 Data Parallelism

Above we describe basic concepts in deep learning models, such as model training and serving. Now we discuss two major distributed paradigms for deep learning models, namely data parallelism (Section 2.2) and model parallelism (Section 2.3). We explain data parallelism in this section.

The data parallel training paradigm is depicted as Figure 2.3. In data parallelism, both machine 1 and machine 2 hold a full copy of the model parameters. In each iteration, they conduct local model training on separate input data, which is shown in Figure 2.3 as input data (partition 1) on machine 1 and input data (partition 2) on machine 2.

After each local training iteration finishes, all the GPUs involved in the same data parallel training job need to conduct model synchronization, which is denoted as the double-headed arrow in Figure 2.3. We illustrate model synchronization with a 4-GPU example, which is shown as Figure 2.4. In data parallel training, after each GPU generating its local gradients (i.e., ∇W^1, ∇W^2, ∇W^3, ∇W^4 in Figure 2.4), it triggers model synchronization to aggregate all the local gradients together as:

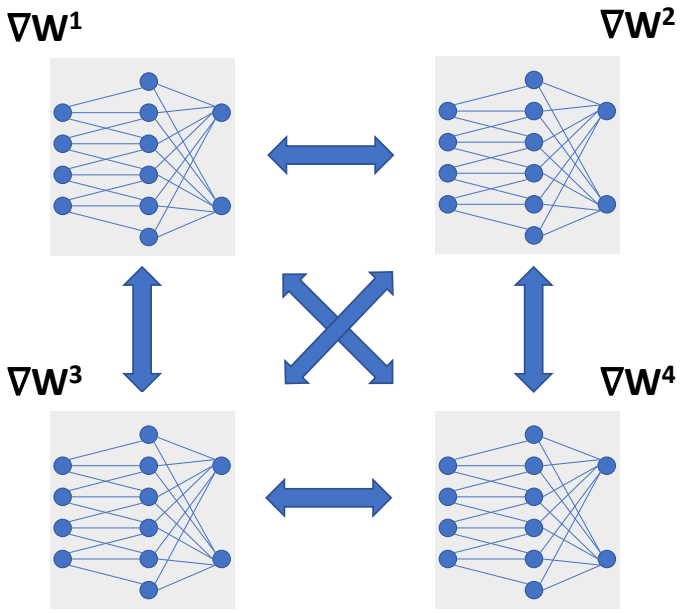


Figure 2.4: Model synchronization in data parallel training.

$$\nabla W = \sum_{i=1}^{n} \nabla W^i \tag{2.3}$$

With the aggregated gradients as ∇W, we broadcast this ∇W to all the GPUs in use, and allow all the GPUs to update their local model parameters based on ∇W.

Model synchronization is the main communication overhead in data parallel training [72]. There are mainly two schemes for model synchronization in data parallel training [73], namely Parameter Server (PS) [24][74][75][76] and Collective Communication [22][31][32][33]. Modern deep learning frameworks [23][29] adopt collective communication as the main scheme for model synchronization, due to its simplicity and scalability [77][78][34][79][80][81]. More specifically, collective communication treats all machines as workers uniformly whereas PS architecture needs to specify two different roles as parameter server and worker. In addition, PS also need to manually assign the ratio between parameter servers and workers among all the nodes involved. And there is no optimal solution on what ratio should be used given arbitrary number of nodes. Thus, in collective communication paradigm, it is much easier to insert or delete worker nodes than parameter server architecture.

Data parallel serving is simpler. Basically, each machine/GPU consumes separate input data partition and generates output predictions in-parallel. No communication is needed during data parallel inference.

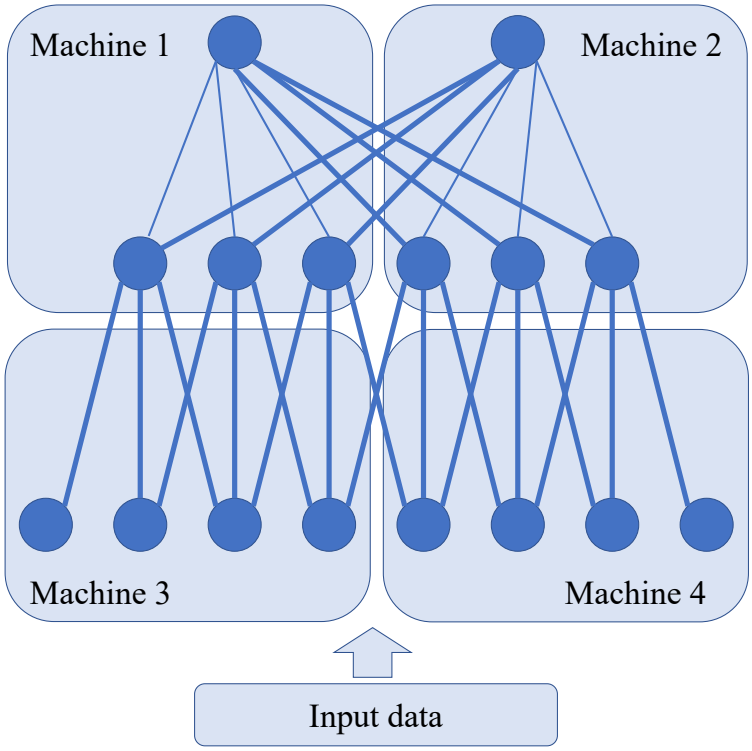


Figure 2.5: Model parallelism paradigm.

2.3 Model Parallelism

In this section, we discuss another distributed deep learning paradigm called model parallelism [82][83][84][28]. Different from data parallelism, in model parallelism, each machine or GPU only holds a portion of the full model parameters.

A 4-node model parallelism setting is shown as Figure 2.5. The model is split into 4 disjoint subsets and each machine holds roughly a quarter of the whole model parameters. As depict in Figure 2.5, all the machines share the same input data during both model training and serving stages.

However, during both model training and serving, they need to transfer lost of intermediate results among each other, which are highlighted as the bold lines across the model partition boarders in Figure 2.5. More specifically, during model parallel training, all the machines need to communication activations during forward propagation and gradients during backward propagation. In model parallel inference, all the machines communication activations in the forward pass only.

In addition, model parallelism naturally creates synchronization barriers due to the sequential dependency of DNN layers [85]. For example, in Figure 2.5, machine 1 and machine 2 cannot start forward propagation until completely receive all the intermediate results from

machine 3 and machine 4. Breaking each input batch in micro-batches and conducting data pipelining over micro-batches [35][36] can mitigate this sequential blocking issue.

Chapter 3

Faster Collective Communication

3.1 Background

As discussed in Section 2.2, collective communication is the main data transfer scheme used in data parallel training [21], which allows GPUs to frequently exchange and synchronize model parameters. It is also used in some modern model parallel training systems like Megatron-LM [28] from Nvidia.

To mitigate collective communication overhead, different companies provide their own solutions, such as Collective Communications Library (NCCL) [31] from Nvidia, Uber's Horovod [32], and Gloo [33] from Facebook/Meta. Incorporated with techniques like wait-free backward propagation that hides communication under computation [86], these collective communication libraries are specially designed to speed-up model synchronization.

3.1.1 Ring-based Collectives

State-of-the-art collective solutions like NCCL and Horovod focus on building rings within the network topology of a single job, and pipeline data transfer over rings they build.

Figure 3.1 shows a toy example of ring-based broadcast in a 4-GPU setting. On the left side of Figure 3.1, it shows a formed ring within the topology. GPU0 want to broadcast its local data to GPU1, GPU2 and GPU3. Before data transfer, GPU0 first partition its local data into small chunks, which are shown as 4 chunks with different colors in Figure 3.1. Assuming the link is unidirectional, GPU0 can pipeline the data transfer over the ring in clockwise direction, which is shown as the right side of Figure 3.1. In reality, all links are bi-directional [87][88][89][90]. Besides this clockwise ring, GPU0 can easily create another ring in counter-clockwise direction using the same group of links in the topology but in the reverse direction.

Ring-based All-Reduce [91][92] follows similar protocol as Ring broadcast described above. The main difference from broadcast is that, current node need to aggregate local results with data received from predecessor before sending it to its successor.

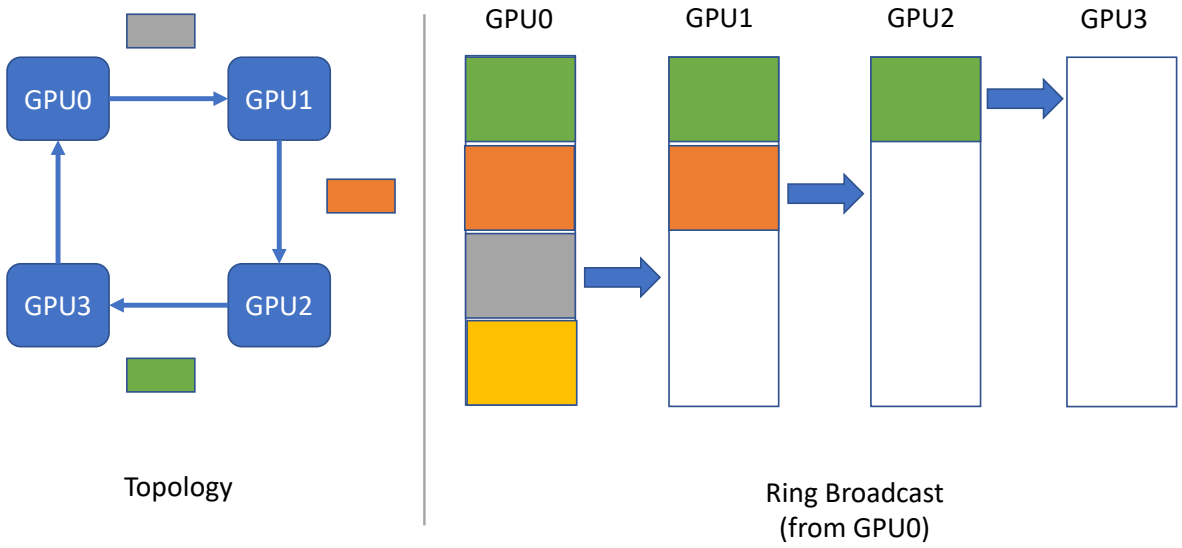


Figure 3.1: Ring-based Broadcast from GPU0.

3.1.2 Topology Heterogeneity

We target on the state-of-the-art multi-GPU hardware with NVLink [88] and NVSwitch [90] like Nvidia's DGX-1 [93] and DGX-2 [94]. We find that, despite incorporate these advanced hardware (e.g., DGX-1 and DGX-2), modern collective communication libraries (e.g., NCCL) still cannot fully mitigate the communication bottleneck in data parallel training. The key issue is ring-based collectives under-utilize hardware links due to topology heterogeneity. We define topology heterogeneity in following three aspects:

First, topology heterogeneity may occur because of different server configurations. As shown in Figure 3.2, for the same DGX-1 8-GPU box, it may have two different versions of network topology. If the DGX-1 server is embedded with P100 GPUs [95], the NVLink topology among GPUs is shown as the left side of Figure 3.2, which is called a hyper-cube topology. Each link here is the first generation NVLink, which provides around 18GB/s. If the DGX-1 box is incorporated with V100 GPUs [96], besides that hyper-cube topology in P100 version DGX-1, there is an additional ring added into the topology of DGX-1 V100 version, which is denoted as red dashed arrow-lines on right side of Figure 3.2. Therefore, in order to fully utilize all the hardware links effectively in different DGX-1 versions, the collective protocols need to be topology-aware.

Second, existing solutions cannot use heterogeneous links within the system. Servers like DGX-1 mainly have two different kinds of interconnects for intra-machine communication. First, it contains point-to-point interconnects called NVLink [88], which is shown as Figure 3.2. NVLink is GPU-exclusive communication links, where each provides 18-25GB/s bi-directional communication bandwidth. DGX-1 box has traditional PCIe bus, which is a

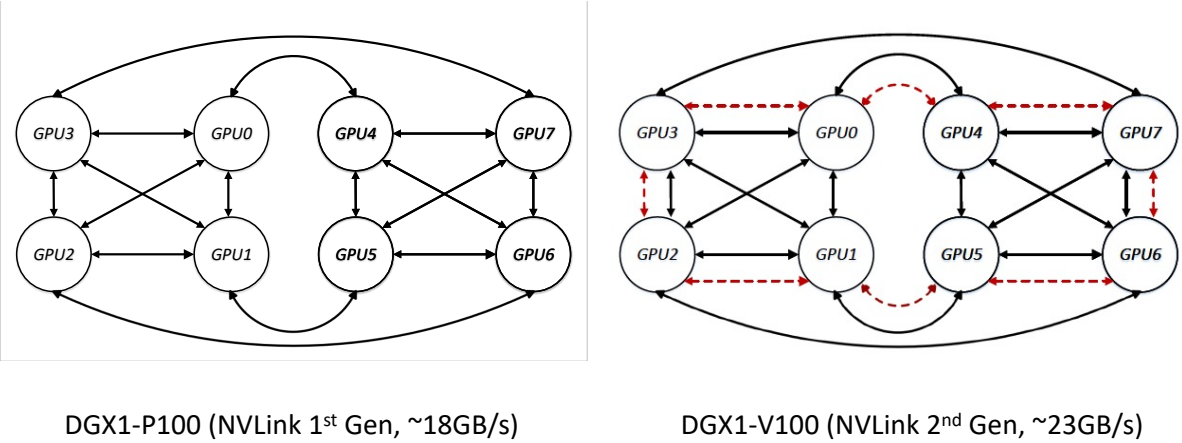
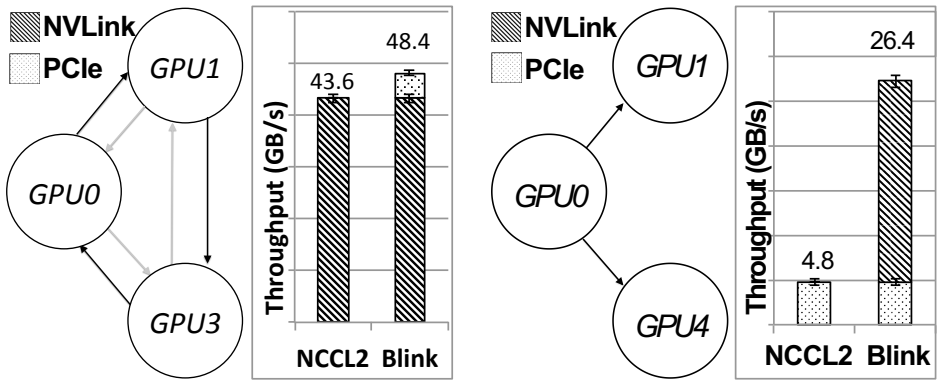


Figure 3.2: NVLink topology of DGX-1 8-GPU server (left with P100 GPUs, right with V100 GPUs).

shared bus between host and devices. PCIe 3.0 [87] can reach peak throughput between 8 to 12 GB/s. PCIe bus connects multiple GPUs together via switch hierarchy within the same DGX-1 machine. Ring-based collectives such as NCCL and Horovord, they fail to use heterogeneous links. The key reason is that, the throughput of a ring is limited by the link with lowest bandwidth. For example, low bandwidth PCIe will be the communication bottleneck if it is included in a high bandwidth NVLink ring. Thus for intra-node communication, Ring-based solution like NCCL prioritizes NVLink over PCIe. Left side of Figure 3.3 shows a 3-GPU broadcast case from GPU0. Since all the GPUs are fully connected with NVLink, NCCL will build two unidirectional rings (one: GPU0–>GPU1–>GPU3–>GPU0, the other: GPU0–>GPU3–>GPU1–>GPU0) over bi-directional NVLink, and abandons PCIe bus completely.

Third, in multi-tenant clusters, the job schedulers are often oblivious to hardware interconnect topologies among GPUs. And multiple jobs can be allocated within the same multi-GPU machine. In addition, topology-aware scheduler must also embrace fragmented allocation to avoid queuing delays [97][98][99]. A simple example of fragmented allocation can be: a 8-GPU job may be assigned with 3 GPUs on 1 machine, and 5 GPUs on another machine. We analyse over 40,000 multi-GPU jobs on a multi-tenant cluster in the cloud. As shown in Figure 3.4, even though ML practitioners always require GPUs in powers of two, it is fairly common that the number of GPUs assigned to jobs can be 3,5,6,7 within each 8-GPU server. Although by adopting scheduling method like Gandiva [38] that is topology-aware and job migration enabled, such schemes has a higher entry barrier since replacing all the independent scheduling frameworks is cumbersome and almost impossible. Furthermore, not every job can be placed and migrated properly given varied job arrival rates.

Ring-based collective protocols may under-utilize links given the above 3 challenges under



(a) Fully connected case. (b) Partially connected case.

Figure 3.3: 3-GPU Broadcast from GPU0 using both Blink and NCCL on DGX-1-P100 server.

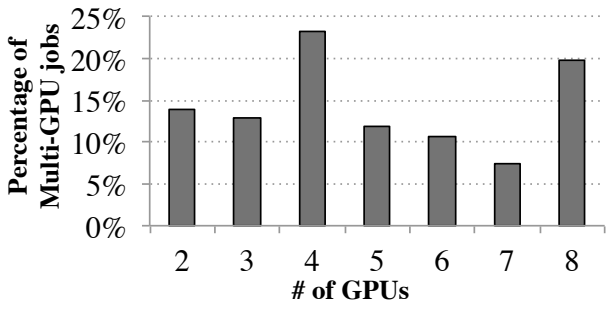


Figure 3.4: Number of GPUs placement for each job within each 8-GPU server on a cloud cluster allocated with 40,000 multi-GPU jobs.

the umbrella of topology heterogeneity. For instance, on right side of Figure 3.3, if a job is assigned with GPU0, GPU1 and GPU4, NVLink ring cannot be formed since there are only 2 NVlinks among these three GPUs (i.e., lack of NVLink between GPU1 and GPU4). In such a case, existing solution like NCCL will fall back to PCIe if it cannot form a NVLink ring.

3.2 Motivation

In this section, we first illustrate why ring-based solution may under-utilize hardware links with several case study. Then we highlight the benefits of packing spanning trees in the face of

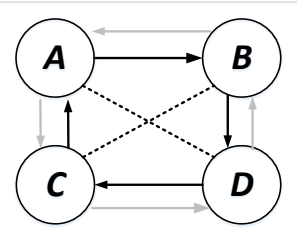


Figure 3.5: Ring-based Broadcast from A (4-node fully connected).

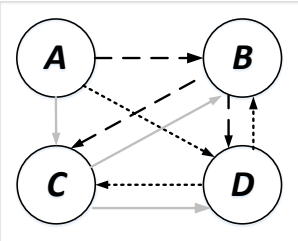


Figure 3.6: Blink Broadcast from A (4-node fully connected).

topology heterogeneity. Last, we present our micro-benchmark results which characterizing the capabilities of modern GPUs (e.g., V100) that helps guide our Blink [34] design.

3.2.1 Trees vs Rings

Our work is motivated by high communication overhead in data parallel training workloads on high-end multi-GPU machine like Nvidia's DGX-1 [93] and DGX-2 [94]. And this high communication overhead still exists even by setting the largest mini-batch size on each GPU and leveraging advanced collectives like NCCL with non-blocking layer-wise backward propagation [86] techniques. The communication overhead becomes more pronounced given increased model sizes and faster computation on newer GPUs like V100 [96], A100 [100] and H100 [101].

More crucially, even within a single DGX-1 machine, the communication overhead is amplified and becomes training bottleneck. The main reason is existing solutions like NCCL or Horovod fail to handle topology heterogeneity we defined in Section 3.1.2. These schemes first build rings given network topology and then pipeline data transfer over the rings. However, ring-based solution have several major structural limitations. First, for each ring, each node can only maintain one input degree and one output degree. This strong structural restriction makes it impossible for ring-based solution to fit into arbitrary and irregular network topologies. Thus it leads to link under-utilization. Second, in order to create a ring, number of links needed is equivalent to the number of GPUs (i.e., N) involved in the same training job. However, the minimum number of links to connect N GPUs is actually $N-1$ links.

Besides the example shown on the right side of Figure 3.3, we provide more toy examples of how rings' structural constrains cause link under-utilization. Figure 3.5 shows a 4-node broadcast setting where node-A broadcast its data to all the other three nodes. These four nodes are fully connected, which is equivalent to left or right four GPUs in Figure 3.2 with P100 setting. Since links are usually bi-directional, node-A forms two rings: one in clockwise direction (i.e., A –>B –>D –>C –>A), the other in counter-clockwise direction (i.e., A –>C –>D –>B –>A). Since there are two concurrent data transfer channels, node-A can split its local data into half and half, then pass first half of data through one ring and the second

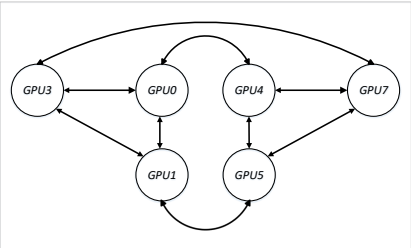


Figure 3.7: 6-GPU topology on DGX-1-P100.

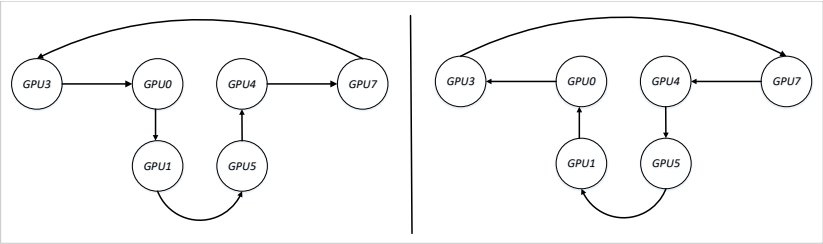


Figure 3.8: NCCL 6-GPU Rings (Broadcast from GPU3).

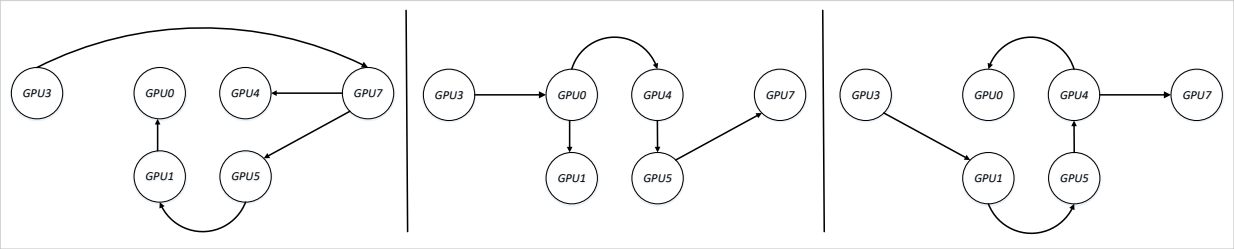


Figure 3.9: Blink 6-GPU spanning trees (Broadcast from GPU3).

half through the other ring. However, the two links in the center are never used during this broadcast process, which are denoted as dashed lines in Figure 3.5. Another more complicated example can be found as Figure 3.8. Figure 3.7 shows the NVLink topology among 6 GPUs inside a DGX-1-P100 machine (i.e., left of Figure 3.2). Here GPU3 initialize broadcast to all the other 5 GPUs. As shown in Figure 3.8, ring-based solution like NCCL also forms two rings, one in clockwise direction and the other in counter-clockwise direction. However, the links between GPU1 <–>GPU3, GPU5 <–>GPU7, GPU0 <–>GPU4 are never used in the 6-GPU broadcast case.

Figure 3.10 and Figure 3.11 show detailed measurements of communication overhead as percentage of end-to-end per-iteration training time. We study over four popular ConvNets as AlexNet [5], VGG [102] and ResNet [6] using NCCL as the communication backend on single DGX-1 machine with P100 GPUs and V100 GPUs. Since a fixed number of GPUs may form different NVLink topology, we measure the best-worst case as a range for each particular number of GPUs in use. As shown in Figure 3.11, the communication overhead from NCCL can be up to 50% of the end-to-end DNN training time on a DGX-1-V100 machine.

In contrast, by modeling hardware links as edges of a graph, classic graph theories [103][104] suggest that, packing spanning trees could lead to maximum network flow from a root node to all other nodes in a directed graph. Therefore, one-to-many and many-to-one primitives such as broadcast and reduce can be directly applied over these uni-directional spanning trees. Furthermore, AllReduce can be split into reduce in one direction then broadcast in

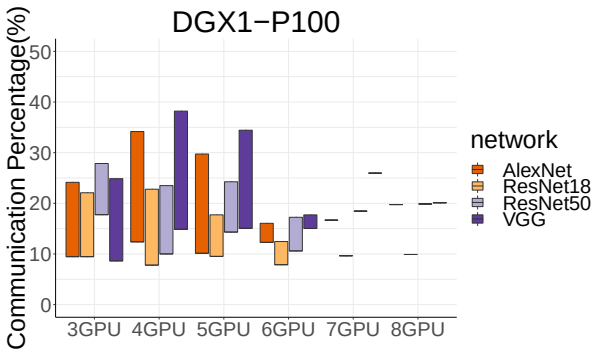


Figure 3.10: Best-worst case of communication percentage regarding to end-to-end DNN training iteration time (DGX-1-P100).

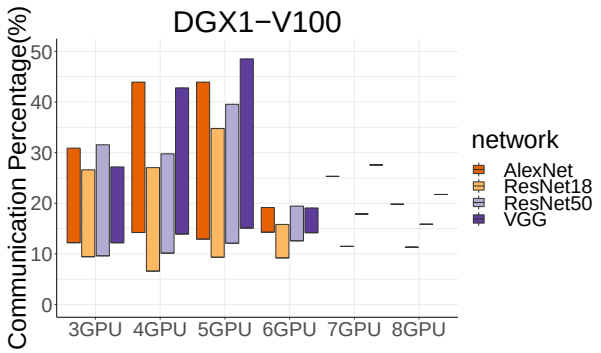


Figure 3.11: Best-worst case of communication percentage regarding to end-to-end DNN training iteration time (DGX-1-V100).

the reverse direction among the spanning trees we have.

By packing spanning trees rather than forming rings, we can achieve optimal link utilization. The advantage of trees over rings are shown in examples of Figure 3.3, Figure 3.6 and Figure 3.9. On the right of Figure 3.3, since spanning trees can be formed with just 2 NVLinks in this setting, our spanning tree solution (i.e., Blink) can leverage the high-bandwidth NVLink for broadcast in this partially connected 3-GPU case. In Figure 3.6, we can pack 3 uni-directional spanning trees from node-A, which utilize all the links available in this 4-GPU setting. In Figure 3.9, we can pack 3 spanning trees with root of GPU3 and use up all the links available in this 6-GPU irregular topology.

With spanning tree paradigm, each GPU could have various degrees of inputs and outputs. Next, we want to evaluate how close to line-rate that GPUs perform using multiple trees not rings.

3.2.2 Micro-benchmarks

In this section, we measure the network throughput among our cases of having computation inline with communication over spanning trees. We present our results on DGX-1-V100 machines. We also conducted measurements over DGX-1-P100 machines, which show similar results. For the sake of brevity, we exclude those results in this thesis. We categorize our measurements into following three settings.

Breadth Test: First, we test fan-in, fan-out data transfer together with inline reduce operations. As shown in Figure 3.12, for fain-in forward case, a central node (i.e., GPU4) collects three input data streams from GPU1,2,3 then forwards towards GPU5. For fan-in reduce and forward defined as Figure 3.13, the central node (i.e., GPU4) aggregates incoming data with its local data by computing a reduction, and then forward aggregated results to its successor (i.e., GPU5). Our fan-out forward defined in Figure 3.14 is the reverse data

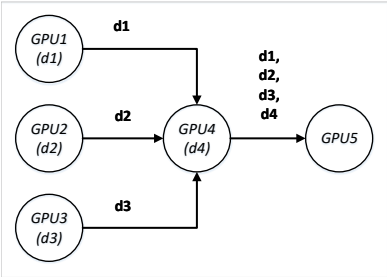


Figure 3.12: Breadth Test: Fan-in forward.

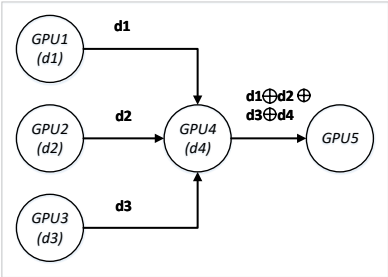


Figure 3.13: Breadth Test: Fan-in reduce and forward.

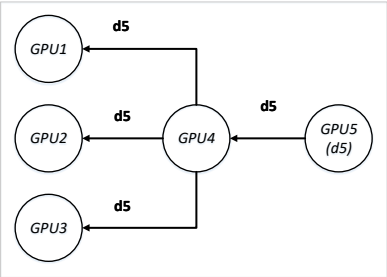


Figure 3.14: Breadth Test: Fan-out forward.

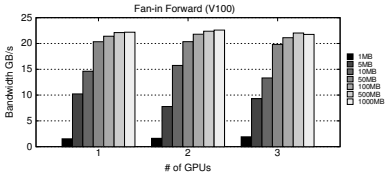


Figure 3.15: Breadth Test Throughput: Fan-in forward.

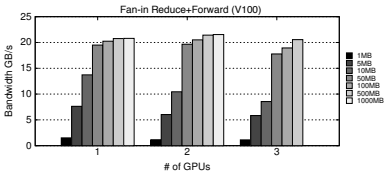


Figure 3.16: Breadth Test Throughput: Fan-in reduce and forward.

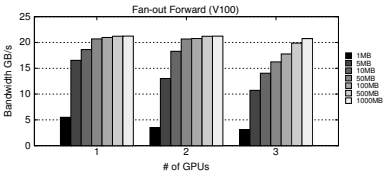


Figure 3.17: Breadth Test Throughput: Fan-out forward.

forward of our fan-in forward. Basically, GPU4 broadcasts GPU5's data to GPU1,2,3.

We test network throughput over above three breadth test cases by transferring data size ranging from 1MB to 1000MB. The corresponding results are shown as Figure 3.16, Figure 3.16 and Figure 3.17 respectively. In summary, with data size equal or larger than 50MB, all of these three breadth test can achieve near maximum network throughput. Compared with fan-in forward, fan-in reduce and forward has 1-2 GB/s lower throughput, which is mainly due to reduction computation latency. Fan-out forward has similar network throughput as fan-in forward, which is close to the line-rate of NVLink.

Depth Test: Next, we conduct our micro-benchmark of depth test, which creates chains for data forwarding and reduction. Here we mainly measure three kinds of chain traffic: forward (Figure 3.18), reduce and forward (Figure 3.19), then reduce and broadcast (Figure 3.20). For chain forward in Figure 3.18, GPU1 is the source with data d1. GPU2 and GPU3 forward the data to GPU4 in the end. For chain reduce and forward in Figure 3.19, for forwarding nodes like GPU2 and GPU3, they aggregates local data with data received from predecessor, and then forward aggregated results to their successors. For chain reduce and broadcast in Figure 3.20, all the nodes execute reduce and forward operation from left to right, then broadcast the final results from right to left.

We also test above three depth test cases with various data size ranging from 1 MB to 1000 MB. The length of chain we tested is ranging from 3-GPU to 8-GPU within an DGX-1-V100 server. For chain forward only results in Figure 3.21, with data size over 500

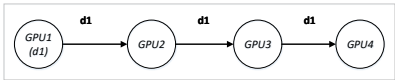


Figure 3.18: Depth Test: chain forward.

Figure 3.19: Depth Test: chain reduce and forward.

Figure 3.20: Depth Test: chain reduce and broadcast.

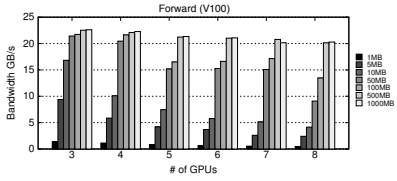


Figure 3.21: Depth Test Throughput: chain forward only.

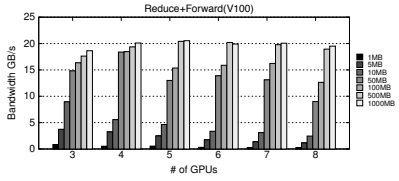


Figure 3.22: Depth Test Throughput: chain reduce and forward.

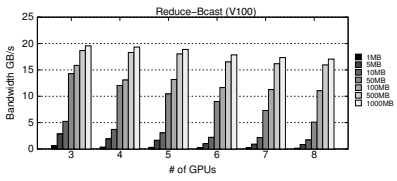


Figure 3.23: Depth Test Throughput: chain reduce and broadcast.

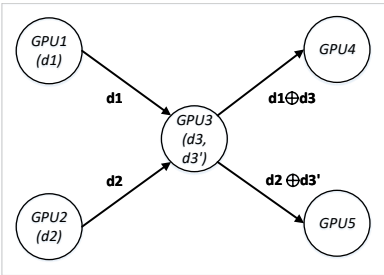


Figure 3.24: Multi-Input, Multi-Output (MIMO).

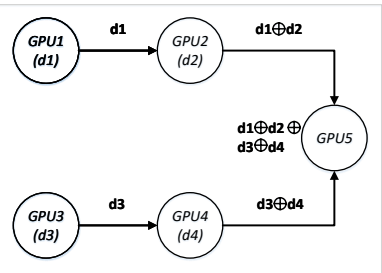


Figure 3.25: Multi-chain aggregation (MCA).

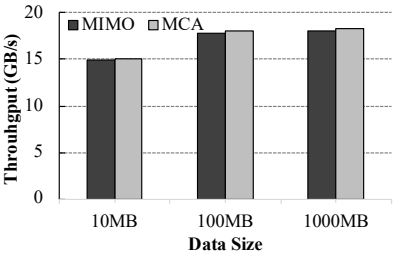


Figure 3.26: MIMO and MCA throughput.

MB, the throughput decreases from 22 GB/s in 3GPU case to around 20 GB/s in 8-GPU case. This pattern is less noticeable in the case of reduce and forward in Figure 3.22. For ore than 500 MB data, the average throughput in chain reduce and forward is around 18 GB/s in Figure 3.22. For chain reduce and broadcast shown as Figure 3.23, with more than 500 MB data, the throughput decreases from around 19 GB/s to around 16 GB/s when we increase the length of the chain from 3 to 8 GPUs. Overall, we see the throughput drops with smaller data sizes. The main reason are twofold. First, it is difficult to fully saturate high bandwidth links with small data sizes. Second, there are constant control overheads for launching CUDA [105] kernels.

Multi-transfer Test: Now we consider the case of concurrent data flow in any given network topology. We mainly test two caces: multiple-input-multiple-output (MIMO) in Figure 3.24 and multiple-chain aggregation (MCA) in Figure 3.25.

In our MIMO configuration in Figure 3.24, GPU1 and GPU2 send their local data (d1, d2) to the center node GPU3. Then GPU3 aggregates its local data d3 and d3' with d1 and

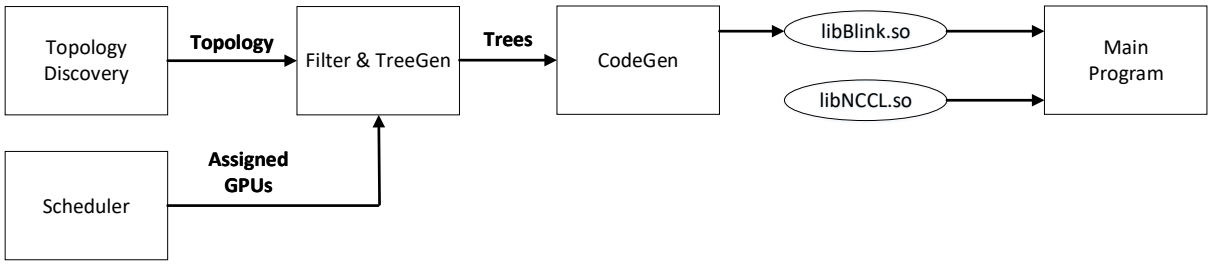


Figure 3.27: Blink workflow.

d2 separately. GPU3 sends d1+d3 to GPU4 and sends d2+d3' to GPU5 simultaneously. In our MCA setting, basically it merges two reduce-and-forward chains together. In the upper chain, GPU2 aggregates local data with GPU1's, then forwards aggregated results to GPU5. GPU4 reduce local data with GPU3's, then forward reduction values to GPU5. GPU5 aggregates both chains' input with its own local data.

The throughput result of MIMO and MCA is shown as Figure 3.26. For smaller data size like 10 MB, both MIMO and MCA can only achieve throughput around 15 GB/s, which is mainly due to the small data chunk cannot fully saturate the NVLink bandwidth. For data size in the range of 100 MB or 1000 MB, both MIMO and MCA can achieve around 18 GB/s throughput.

Summary: From all the micro-benchmark results above, we see that modern GPUs like V100 with NVLink interconnects could provide good support for data transfer over spanning trees with various depth and width. We also see the throughput drops a little when incorporating computation kernels like reduction with communication. In summary, these micro-benchmarks make it promising and possible for using spanning trees to implement collective communications such as broadcast and allreduce.

3.3 Blink Design

In this section, first we provide a brief system workflow of our Blink project. We describe Blink design and our techniques to handle heterogeneous links while achieve high link utilization. Second, we illustrate how we use uni-directional spanning trees for one-to-many and many-to-one primitives such as reduce and broadcast. Third, we also extend it into many-to-many primitives such as allreduce by leveraging the bi-directional hardware interconnects. Fourth, we provide solutions to use PCIe and NVLink together for collective communication. Last, we extend our method to DGX-2 [94] and multi-GPU multi-machine cases.

3.3.1 System Overview

We provide the overview of Blink project in this section. Blink is a fast and generic collective communication library. The workflow of Blink is shown as Figure 3.27.

Given arbitrary network topology, Blink dynamically generates the optimal collective communication scheme by packing maximum number of non-conflicting spanning trees in the topology. Now we decribe the major components and workflow of Blink project in Figure 3.27. The workflow of Blink is composed of the followings:

- At job runtime, after the scheduler assigns a set of GPUs to the DNN training job, Blink probes the network topology and available links among these assigned GPUs.
- Once Blink collects the network topology information, we pass this topology information into our TreeGen module. Our TreeGen computes the maximum fractional packing of spanning trees in the given topology. We use these spanning trees for the implementation of our collective communication protocol.
- We then pass the spanning trees from TreeGen to our CodeGen, which generates the code for data transfer commands over these spanning trees. The code generated provides same API as NCCL, and is packed as a shared library (i.e., libBlink.so in Figure 3.27).
- Last, we use LD_PRELOAD flag to dynamically load our Blink implementation to replace the NCCL counterpart (i.e., libNCCL.so in Figure 3.27) when the main program is invoked.

We next illustrate each module in Figure 3.27 in the following sections.

3.3.2 Packing Spanning Trees

We first consider one-to-many primitives such as broadcast using uni-directional spanning trees. We model the network topology information as follows. We treat each GPU as a vertex V and each hardware link as a directed edge E. Each directed edge is assigned with a bandwidth value which is proportional to its link capacity. Thus, these vertices and edges creates a directional graph.

With the above graph model, the optimal throughput for broadcast is to find the maximum weights of flows that start from a given root node and reach all other vertices in the graph. This problem is well-studied in classic graph theory [104]. Basically, the objective function is to find the maximal packing of uni-directional spanning trees or arborescences in a directed graph [103]. Thus, we formalize our optimization problem as follows:

$$\max \sum_i w_i \tag{3.1}$$

$$\text{such that } \forall e \in E, \sum_i \kappa_i * w_i < c_e \tag{3.2}$$

$$\text{where } \kappa_i = \begin{cases} 1, & \text{if } e \in T_i \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

Formally speaking, our objective is that, given a directed graph G with edges E , root vertex r, vertices V and set of spanning trees T, we want to find the weights w_i so that the sum of w_i trees passing through edges in E will not exceed the capacity of any edges.

However, the search space of above formalized problem too large. For example, in a complete graph, the number of arborescences can be at the exponential scale as $\mathcal{O}(n^{n-2})$. Therefore, the runtime for problem solving can be insanely long. Some exact and more computational efficient algorithm has been proposed recently, which reduces the computational complexity to $O(n^3 mlog(n^2/m))$ with m edges and n vertices [106]. However, we still abandon this exact and slightly efficient solution due to its high computational complexity. We adopt an approximate algorithm in the next section.

3.3.3 Approximate Tree-Packing

We adopt an approximation technique called multiplicative weight update (MWU), which is usually used in optimization in game theory. Our use of MWU follows a recent work to achieve near-linear time approximation in fractional packing problems [107]. For instance, it can solve spanning tree packing problem in $O(m \ln m/\epsilon^2)$ where m is number of edges in the graph.

This MWU-based approach for packing spanning trees works as follows: we initialize each edge in E with capacity value and another weight indicating how much the capacity of each edge has been used. Given this setting, we can use iterative method such that, during each iteration, we find the minimum weight spanning tree in current assignment. We increase the weight by a factor of ϵ and then update the whole graph accordingly. Therefore, this algorithm can be converged after $O(\ln m/\epsilon^2)$ iterations. Upon the convergence, we can get a set of spanning trees $\{T_i, i \in \mathbb{N}\}$ and corresponding weights $\{w_i, i \in \mathbb{N}\}$. The total bandwidth B for broadcast is as the sum of weights as follows:

$$B = \sum_{i \in \mathbb{N}} w_i \tag{3.4}$$

However, this MWU-based algorithm [107] has no bound on the number of spanning trees it generates. After solving this MWU optimization problem, we find that the number of arborescences generated can be too large. For example, the number of arborescences

generated is 181 within an 8-GPU server. Thus, it is impossible to use so many spanning trees together for concurrent data transfer. It is because the data size per each tree will be too small to fully saturate the link bandwidth.

To minimize the number of spanning trees being generated, we formulate an integer linear program (ILP). Here we limit the weights w_i can only be 0 or 1. Thus, the whole problem can be formalize as below:

$$\max \sum_{i=1}^{k} w_i \tag{3.5}$$

$$\text{such that } \forall e \in E, \sum_{i} \kappa_i * w_i < c_e \tag{3.6}$$

$$\forall w_i \in \{0, 1\} \tag{3.7}$$

$$\text{where } \kappa_i = \begin{cases} 1, & \text{if } e \in T_i \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

Here k can be controlled by the number of arborescences generated from previous MWU procedure. Solving this ILP will generate $\hat{c}$, which may be lower than theoretical optimal as c^*. We then iteratively relax constrains until the difference between $\hat{c}$ and c^* is under some threshold (e.g., 3%). With this integer approximation, we can reduce the number of trees from 181 to 6 in an 8-GPU DGX-1 server. With total data size as 1000 MB, now each tree is responsible for transferring around 166 MB data which could fully saturate link bandwidth.

3.3.4 Extending to Many-to-many Collectives

Above we discuss one-to-many or many-to-one collective primitives such as Broadcast, Gather, Reduce. In this section, we further extend it to many-to-many primitives such as AllGather and AllReduce.

We exploit the fact that all the modern hardware links are bi-directional. Thus, for each uni-directional spanning tree we have, we can run many-to-one primitive in one direction and run one-to-many primitive in the reverse direction. Taking AllReduce as an example, in Figure 3.20, we pick the root node as GPU4 on the right. For AllReduce in this 4-GPU setting, we reduce from left to right, then broadcast from right left. Thus it finishes the AllReduce operation.

This stratgy of combing two unidirectional spanning trees also guarantee we match the lower bound of number of messages passing for many-to-many primitives (e.g., AllReduce). For example, in AllReduce operation, our solution achieves the lower bound of $2 \times \lceil \frac{N-1}{N} \rceil$ as shown in previous literature [108].

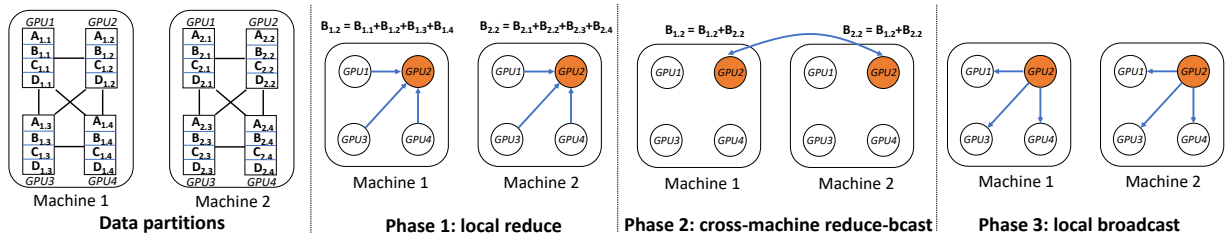


Figure 3.28: Blink's Three-phase AllReduce protocol for cross-machine settings.

3.3.5 Hybrid Communication

We provide concurrent data transfer over both PCIe and NVLink for intra-machine collective communication. The first challenge we faced is, there is no official way to switch between NVLink and PCIe for communication. In our experience, we find by using cudaDeviceDisablePeerAccess(), it can force data transfer on PCIe and disable NVLink communication.

Another challenge is how to balance data transfer over PCIe and NVLink such that they can finish at roughly the same time. We denote D_{total} as the total amount of data needs to be transferred. D_{NVL} and D_{PCIe} are the data splits between NVLink and PCIe bus. T_{DPA} refers to the latency for calling cudaDeviceDisablePeerAccess() function. We denote the bandwidth of PCIe and NVLink as BW_{PCIe} and BW_{NVL}

$$\begin{aligned} &\text{Objective Fn: } T_{PCIe} + T_{dpa} = T_{NVL} \\ \implies D_{PCIe} =& \frac{D_{total} \times BW_{PCIe}}{BW_{PCIe} + BW_{NVL}} - \\ & \frac{T_{dpa} \times BW_{PCIe} \times BW_{NVL}}{BW_{PCIe} + BW_{NVL}} \\ D_{NVL} =& D_{total} - D_{PCIe} \end{aligned} \tag{3.9}$$

Given these notation and objective function in Equation 3.9, we can find the optimal data split between NVLink and PCIe. T_{DPA} is empirically measured and may vary given different number of GPUs in use. And we measure this value on the first few initial calls into our library.

3.3.6 DGX-2 and Multi-machine Settings

In this section, we first describe our collective scheme in DGX-2 [94] with NVSwitch [90]. Then we illustrate our multi-machine design.

DGX-2: DGX-2 has 16 V100 GPUs inside a single box. These GPUs are connected over NVSwitch. Each GPU is connected with 6x NVLink bandwidth, which is equivalent to around 130 GB/s. For DGX-2 machine embedded with NVSwitch, the connectivity among any subset of the GPUs in DGX-2 is now uniformed. In DGX-2, NCCL constructs double-

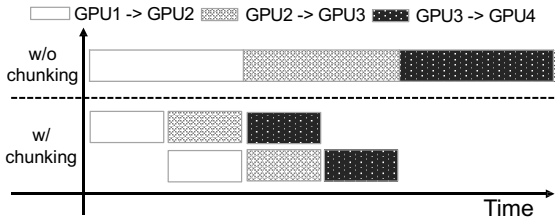


Figure 3.29: Chunking data to reduce multi-hop network latency.

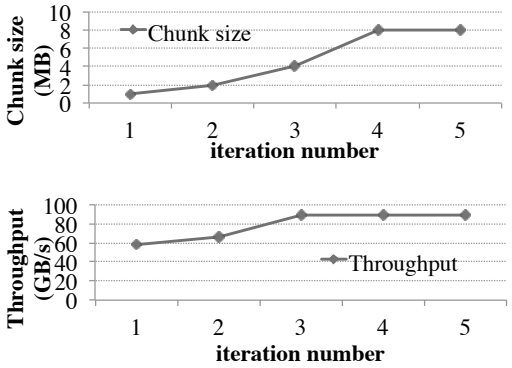


Figure 3.30: Automatic chunk size selection using multiple-increase additive-decrease (MIAD).

binary trees [109] with data size smaller than 16 KB, and still forms rings for collective communication with larger data size.

In contrast, Blink still maintains spanning tree scheme. More specifically, we create single-hop trees which can be direct support for one-to-many and many-to-one primitives such as Gather, Broadcast. For many-to-many primitives like AllReduce, we still apply reduce in one direction and broadcast in the reverse direction over the same spanning tree. Given n GPUs, we create n such one-hop tree. Each GPU acts as a root node for data collection of $\frac{1}{n}$ total data. Our one-hop tree solution guarantees we have the minimum end-to-end network latency.

Multi-machine: In multi-machine case, Blink provide a 3 phase protocol, which is shown as Figure 3.28. As shown in Figure 3.28, Data item $X_{m.g}$ denotes data partition X on server m's GPU g. For each data partition, it maintains a distinct server-to-local root. For the sake of simplicity, Figure 3.28 only shows the reduction (function is denoted as symbol $+$) for partition B and the root is at $GPU2$. Similar protocol can be applied for the other data partitions.

This 3-step protocol works as follows:

- Intra-server local reduce: In each machine, the root of each tree aggregates data from its children nodes.
- Cross-machine reduce and broadcast: It is similar as the single-hop trees in DGX-2 machine. For n servers, we create n such one-hop cross-machine trees, where each tree is for data collection of $1/n$ total data.
- Intra-server local broadcast: at this stage, each server's local roots broadcast the results obtained in the second phase above to all the nodes inside the machine.

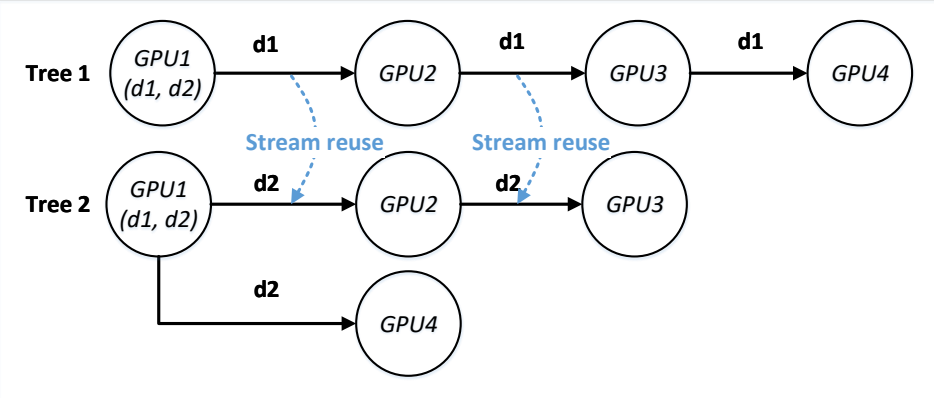


Figure 3.31: Stream reuse for fair sharing of links.

3.4 Implementation

This section describes our CodeGen module. Basically, CodeGen translates the spanning trees generated from TreeGen into real data transfer commands. Here we implement two optimization components to achieve ideal system performance.

3.4.1 Automatic Chunk Size Selection

In each CUDA stream, data chunk is the atomic and smallest unit for data transfer over our spanning trees. One common way to reduce end-to-end network latency is to chunk data into smaller pieces and pipeline the data transfer.

As shown in Figure 3.29, by chunking the data into half and half and pipelining data forwarding, we can reduce the amount of time which is similar as the length of black bar in Figure 3.29. Our objective is to parallelize or pipeline data transfer with minimum network latency. Intuitively, smaller chunk size should always lead towards lower end-to-end latency. However, if the chunk size is too small, it may not be able to fully saturate the link bandwidth and also introduces too much system scheduling overheads.

To select proper chunk size on each CUDA stream, we adopt an adaptive scheme from TCP/IP Stack [110], which is multiplicative increase and additive decrease (MIAD). We start with a small size and increase the chunk size by multiplicative factor when the corresponding throughput is increasing. On the other hand, if the throughput decreases, we additively decrease our chunk size to find an steady and near-optimal state.

As shown in Figure 3.30, we start with small chunk size as 1 MB in our first iteration and measure the throughput. We then double the chunk size in iteration 2,3,4. Once we reach a steady state, we fix the chunk size (i.e., 8 MB in Figure 3.30) and use this fixed chunk size for later on data transfer.

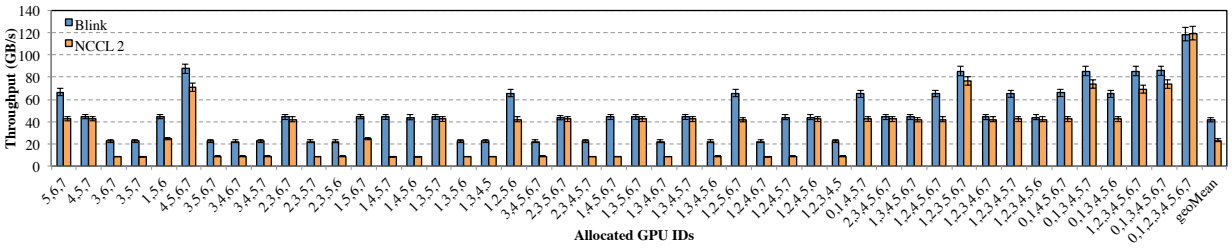


Figure 3.32: Broadcast throughput comparison between Blink and NCCL for all unique topologies in DGX-1-V100 machine.

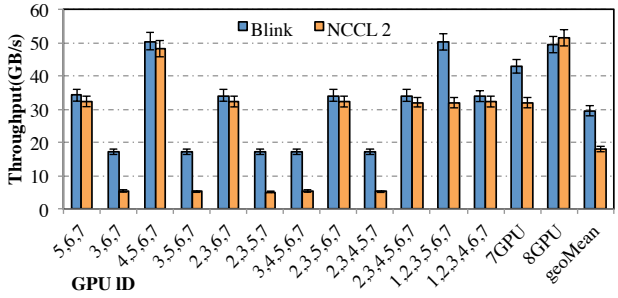


Figure 3.33: Broadcast throughput comparison between Blink and NCCL for all unique topologies in DGX-1-P100 machine.

3.4.2 Link Sharing

Another challenge with our multiple trees scheme is that CUDA does not provide direct control over shared links. For example, if we have two trees passing the same hardware link and each with weight of 0.5, a fair sharing scheme should transmit one data chunk from the first tree followed by one chunk from the second tree, so on and so forth. However, in practice, our CUDA implementation does not allow such fine-grained control over the order of different data chunks. It leads to the cases that chunks from some trees are arbitrarily delayed and harms our overall throughput performance.

Based on our observation, the orders of chunks within the same stream is guaranteed. Thus, we address this issue by reuse the CUDA streams in the case that same link is used by multiple trees at roughly the same position. For instance, in Figure 3.31, we do broadcast from GPU1, which has two data chunk d1 and d2. We create two spanning trees (Tree 1 and Tree 2) from root node GPU1. Note that the link between GPU1 <–>GPU2 maintains the same position for both Tree 1 and Tree 2, instead of initializing a new CUDA stream for Tree 2, we re-use the CUDA stream from Tree 1 to guarantee the fair sharing of the link.

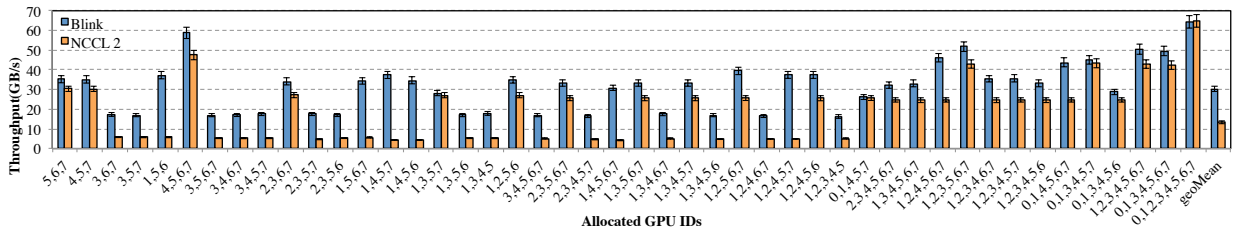


Figure 3.34: AllReduce throughput comparison between Blink and NCCL for all unique topologies in DGX-1-V100 machine.

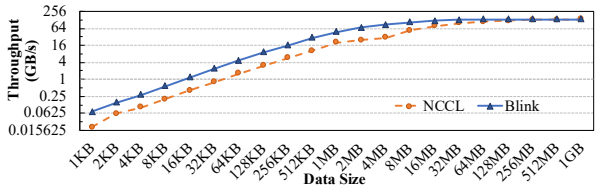


Figure 3.35: AllReduce throughput comparison between Blink and NCCL on a 16-GPU DGX-2 machine.

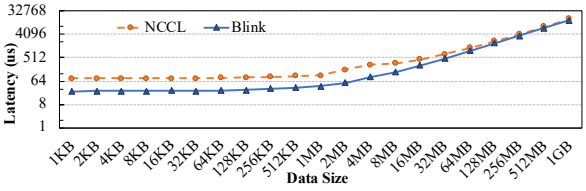


Figure 3.36: AllReduce latency comparison (in μs) between Blink and NCCL on a 16-GPU DGX-2 machine.

3.5 Evaluation

In this section, we compare throughput performance between NCCL 2.4 [31] and Blink for collective communication such as Broadcast and AllReduce. We evaluate on both DGX-1-P100, DGX-1-V100 machine and DGX-2 machine. We also provide the end-to-end speedups of using Blink with 4 popular DNNs in both single machine and multiple machine cases.

3.5.1 Broadcast and AllReduce Micro-benchmarks

Here we compare performance between Blink and NCCL 2.4 on Broadcast adn AllReduce operations. Given the topologies shown in Figure 3.2, we may have different topologies with same or different number of GPUs in use. In total, we have 46 different configurations for different number of GPUs in use and their locations in DGX-1-V100 machine, and 14 different topologies for DGX-1-P100 machine. Figure 3.32 and Figure 3.33 show the Broadcast comparison between NCCL and Blink on DGX-1-V100 machine and DGX-1-P100 machine, respectively. And Figure 3.34 shows AllReduce comparison between Blink and NCCL on a single DGX-1-V100 machine. For Figure 3.32, Figure 3.33, Figure 3.34, the number array on x-axis refers to the allocated GPU IDs in each configuration, which can be directly mapped to Figure 3.2.

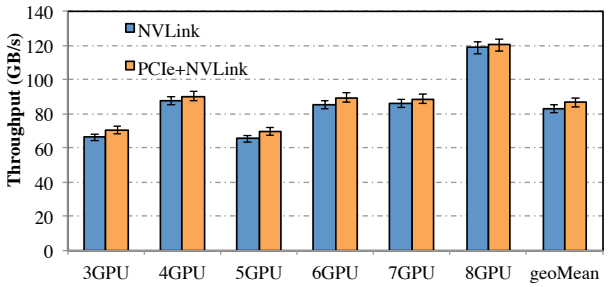


Figure 3.37: Broadcast throughput comparison between hybrid and NVLink-only with various number of GPUs on DGX-1-V100 machine.

3.5.1.1 NVLink Broadcast

Here we provide detailed evaluation for Broadcast comparison between NCCL 2.4 and Blink on both DGX-1-V100 machine and DGX-1-P100 machine.

In Figure 3.32, compared with NCCL, Blink can achieve up to 6x speedup with 2x geo-mean on a DGX-1-V100 machine. In the cases of irregular topology like 1,4,5,6, since NVLink cannot form a ring, NCCL fall back to PCIe for collective communication, which leads to dramatically lower network throughput when comparing with Blink. Even in the cases that NCCL can form rings, Blink still outperforms over NCCL due to our higher link utilization rate and optimized chunk size selection.

For DGX-1-P100 machine results in Figure 3.33, Blink achieves up to 3x speedup with 1.6x geo-mean over NCCL. All the 14 configurations in this DGX-1-P100 machine show similar results as DGX-1-V100 machine in Figure 3.32.

3.5.1.2 NVLink AllReduce

Figure 3.34 lists the AllReduce comparison between NCCL and Blink on a DGX-1-V100 machine with 46 different topology configurations. Generally, AllReduce throughput is lower compared with Broadcast in Figure 3.32. The main reason is computation kernels (e.g., Reduce) are involved in the procedure, thus it may decrease both throughput and latency performance when comparing with Broadcast cases.

Overall, Blink outperforms over NCCL with up to 8x speed-up with 2x geo-mean in throughput. For the sake of brevity, we exclude AllReduce result fro DGX-1-P100 since it closely match the findings as DGX-1-V100 results in Figure 3.34.

3.5.1.3 NVSwitch AllReduce

For DGX-2 with NVSwitch, we direct conduct a 16-GPU AllReduce operation and measure the throughput (Figure 3.35) and latency (Figure 3.36) results. The y-axis in both Fig-

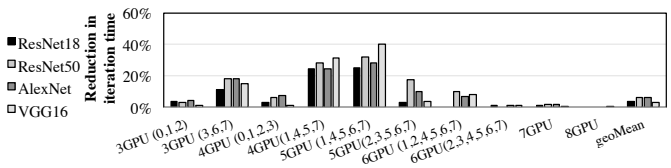


Figure 3.38: Blink training time reduction for each iteration over 4 popular DNNs on ImageNet-1K dataset.

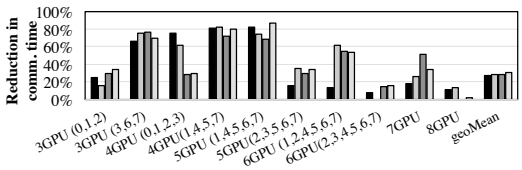


Figure 3.39: Blink communication time reduction over each iteration on ImageNet-1K dataset.

ure 3.35 and Figure 3.36 are log-scaled. We measure throughput and latency with data size ranging from 1KB to 1GB, which are shown as ticks on x-axis of these two figures.

In both Figure 3.35 and Figure 3.36, comparing with NCCL's double binary trees and rings, Blink's single-hop trees (described in Section 3.3.6) achieves up to 3.5x speed-up in throughput and 3.32x latency reduction. Our biggest wins are with small data sizes. This is mainly due to our single-hop trees guarantee the minimum network latency.

3.5.2 Hybrid Transfer

Figure 3.37 shows the evaluation results of our hybrid (PCIe+NVLink) data transfer and NVLink-only in Broadcast on a DGX-1-V100 machine. We tested over the number of GPUs ranging from 3 to 8. Overall, by adopting hybrid data transfer, we can achieve around 2-5 GB/s throughput boost when comparing with NVLink only solution.

3.5.3 End-to-end DNN Training

We test our end-to-end DNN training performance with popular DNNs using PyTorch [29]. The DNNs we use are ConvNets as AlexNet [5], two ResNet [6] and VGG16 [102]. We train these ConvNets on ImageNet-1K dataset [15] and measure our communication time and training iteration time using both NCCL and Blink. We conduct our measurements with both single DGX-1-V100 machine case and multiple DGX-1 machine case.

3.5.3.1 Single Machine

The evaluation results for single machine are shown as Figure 3.38 and Figure 3.39. Here we also test DNN training performance over 3 to 8 GPUs on a DGX-1-V100 server.

As communication reduction results shown in Figure 3.39, switching from NCCL to Blink leads to up to 87% communication time (with 31% geo-mean) reduction in DNN training jobs. In Figure 3.38, Blink outperforms NCCL by up to 40% time reduction (with 6.3% geo-mean) in end-to-end DNN training jobs.

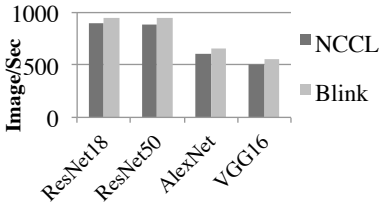


Figure 3.40: System throughput (Image/Sec) for distributed DNN training using two DGX-1-V100 machines.

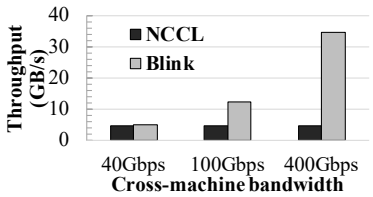


Figure 3.41: AllReduce throughput projections given potential high bandwidth cross-machine interconnects

3.5.3.2 Multiple Machine

As described in Section 3.3.6, Blink's multi-machine collective communication is a three phase protocol. As shown in Figure 3.40, Blink outperforms over Horovod [32] with NCCL and MPI [91] when training four different DNNs over two DGX-1-V100 machines. The reason for this minor improvement is due to limited cross-machine bandwidth. In our test setting, the cross-machine AllReduce throughput is only 40Gbps (i.e., 5GB/s), which is much lower than our intra-machine AllReduce with NVLink (40GB/s).

To project how faster cross-machine interconnects may change our performance, we simulate with different cross-machine bandwidth and measure AllReduce throughput, which is shown as Figure 3.41. The data size we use here is 100 MB. As depicted in Figure 3.41, with high bandwidth cross-machine interconnects [111][112][113], Blink's performance gain will be more pronounced when comparing with NCCL. For example, we can achieve up to 7x AllReduce speedup over NCCL if we have 400Gbps cross-machine bandwidth.

3.6 Related Work

Previous work on collective communication mainly falls into the following two categories.

Topology-fixed Schemes: For fixed topology setting, previous standard collective communication is MPI [91]. Previous literature mainly focus on regular network topology such as full-mesh [114][115] and hyper-cube [116][117][118]. Some more recent work provides optimizations over MPI, such as handling the case that number of nodes is not powers of two [119], or buffer size auto-tune for specific hardware architecture [120].

One line of work is called butterfly algorithm [121][122][123], which achieves lantecy-optimal AllReduce. Butterfly AllReduce is divided into to phases: first is a recursive ReduceScatter then followed by a recursive AllGather. However, the communication patterns in butterfly algorithm can cause network contention very often, thus makes it less practical. Ring-based collectives are shown to be bandwidth optimal in homogeneous network environ-

ments [108][124]. Different companies implement ring-based solution as their own collective communication library such as NCCL [31] from Nvidia, Horovod [32] from Uber, Gloo [33] from Facebook/Meta and IBM Power AI DDL [80]. However, all the literature mentioned above assume to have fixed and regular network topology, which may not be a good fit when topology changes in cloud environments. Blink is designed to handle both irregular and dynamic topology and provides optimal solutions.

Topology-aware Protocols: Topology-aware works are main found in wide-area networks to minimize data transfer over slow links [125][126]. Similar idea has been applied to cloud environments which determine node locality based on pair-wise bandwidth [127]. This method is extended into NUMA multire-core architecture [128]. Recent work like Blueconnect [79] decouples AllReduce into ReduceScatter followed by AllGather. However, it can only be applied to symmetric topologies, which makes it less flexible when comparing with Blink. Blink can be generally applied to both symmetric and asymmetric network topologies, and can also combine heterogeneous links for concurrent data transfers.

3.7 Summary

This part of the thesis provides a fast and generic collective communication library called Blink. To handle the topology heterogeneity, Blink dynamically packing maximum number of spanning tree in the given network topology and achieve near-optimal network throughput. Compared with state-of-the-are ring-based solution like NCCL, Blink achieves up to 8x faster AllReduce, and reduce end-to-end DNN training time by up to 40%.

Chapter 4

Eliminating Communication in Model Parallelism

4.1 Background

Convolution Neural Networks (CNNs) is a specialized kind of DNNs, which enables computers to excel on a lot of vision learning tasks such as image classification [5][6][129], semantic segmentation [52] and object detection [53].

In the past decade, we see the trends that both input image resolutions and model sizes are growing drastically. As mentioned in Section 1.1, the CNNs model size almost increases 200 times over the years, and the number of pixels per image also grows by orders-of-magnitude. To reduce model serving latency, distributed model serving is widely-adopted [23][22][21][130], which allows a single CNN to run in-parallel over multiple machines or accelerators. There are mainly two types of conventional parallelism approaches, namely data parallelism [131][24] and model parallelism [26] [20]. In data parallel serving, each machine or GPU hold the full copy of model parameters and does model inference independently on separate subset of the whole input data. In model parallel serving, each GPU only maintain a portion of the whole CNN model. Lots of intermediate results such as activations are transmitted during in-parallel model serving.

Recently, making faster decision on live data becomes more and more important. For instance, in autonomous driving application scenarios [132][133], if the perception camera captures a frame of image than contains pedestrians ahead, it may save people lives if we can make the stop decision slightly quicker. Similar latency-driven cases can also be found in finance domain. For example, giant banks such as Goldman Sachs [134][135] and J.P. Morgan [136] are adopting deep learning methods for automatic stock trading. In this case, if one party can make the trading decision slightly faster (e.g., several milliseconds earlier) than the other parties, it can bring-in huge amount of profit to the company. From a system point of view, making faster decision on live data is equivalent to making faster decision on each atomic, incoming data item (e.g., instantaneous price of a single stock, each incoming

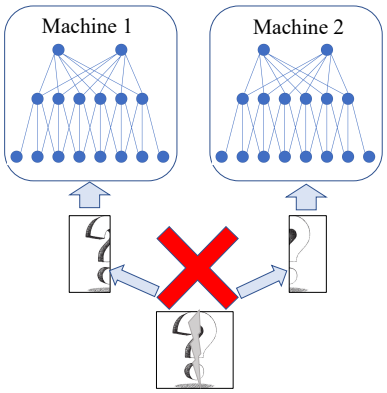


Figure 4.1: Data Parallel model serving.

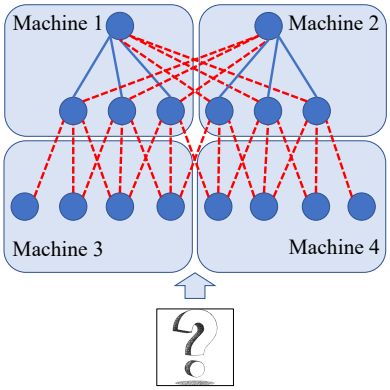


Figure 4.2: Model Parallel model serving.

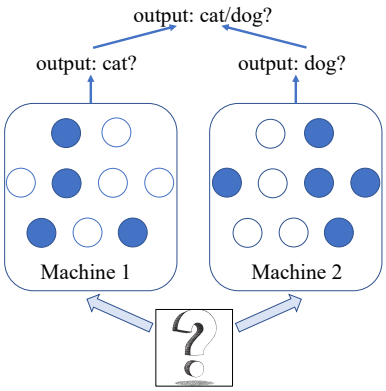


Figure 4.3: Class Parallel model serving (sensAI).

image).

In this case, neither model parallelism nor data parallelism can reduce live data serving latency. As shown in Figure 4.1, it is impossible to split an atomic input item into pieces and do data parallel serving. In model parallel serving (Figure 4.2), all the machines can share the same input piece. However, they need to communicate intermediate results (e.g., feature-maps as red dashed lines in Figure 4.2) among each other for each single input item, which is huge communication overheads.

To achieve low-latency inference on single input piece, we propose sensAI, which is a generic scheme to decouple a big CNN into a bunch of disconnected subnets. This divide-and-conquer scheme only needs negligible communication (i.e., one float value per subnet) and achieves decent inference accuracy. At high level, sensAI achieves near-zero communication overhead by adopting a new concept called Class Parallelism. As shown in Figure 4.3, we decouple a multi-way classification base model into a number of binary classification subnets, where each subnet is only responsible for decision making of a single class (e.g., car or not cat, dog or not dog). The theoretical intuition behind our class parallelism is, different neurons (i.e., channels) are responsible for predicting separate classes of images. Normally, only a subset of channels are crucial for predicting certain class of images [137]. Furthermore, our class parallelism method is orthogonal to data parallelism, which means it can be applied together with data parallelism.

For classification tasks with small number of classes say N (e.g., CIFAR-10 [14]), our class parallelism can be achieved by pulling out N binary classifiers from the base model. And we use these N binary models in-parallel to determine the predicted class by taking the maximum confidence value among these binary classifiers. For classification problem with too many classes (such as ImageNet-1K [7]), we decouple the base model into multiple grouped classifiers (e.g., k groups), where each grouped classifier is a m-way classifier pulled

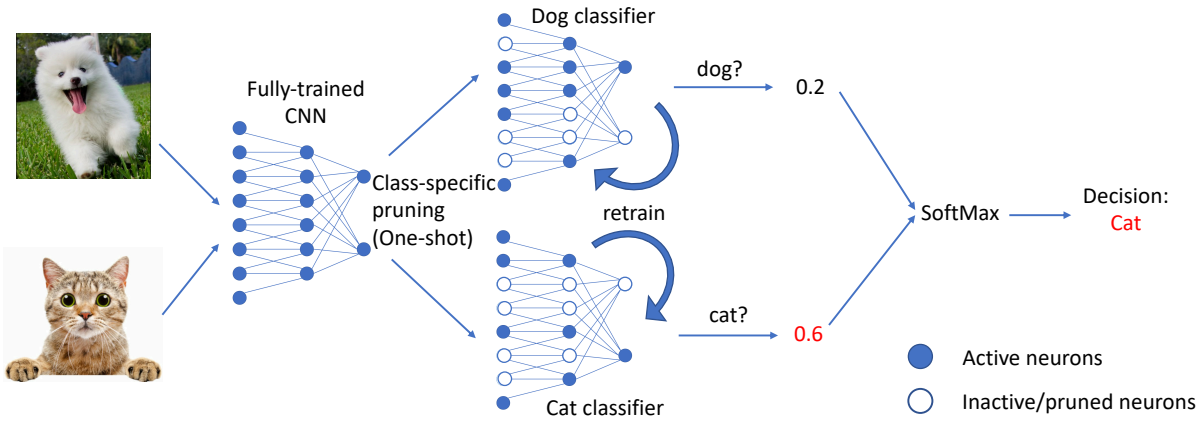


Figure 4.4: sensAI's three-phase workflow for class-parallel inference.

from base model. To guarantee the full class coverage, we ensure that

$$k \times m = N \tag{4.1}$$

This guarantees that using our k m-way classifiers together can cover the original N-way classification problem. Next we will discuss related literature.

4.2 Related Work

The related work to our sensAI project in mainly in following four categories.

Data and model parallelism: To speed up model inference, in-parallel paradigms, such as data parallelism [21][73][34], model parallelism [20][35][27] and hybrid of two [130], have been widely adopted. However, all of these fall short of reducing latency with single atomic input item. As mentioned in Section 4.1, we cannot split atomic input piece for data parallel serving, and model parallel serving introduce huge communication overheads. Different from data and model parallelism, sensAI decouples a base model into disconnected subnets with Class Parallelism, thus the communication in-between can almost be eliminated at the in-parallel model serving stage.

Class-specific neuron analysis: Previous literature provides unit ablation study and shows that unit ablation over a fully-trained CNN only decreases serving accuracy of certain classes [138]. Recent work also shows the possibility of decouple a 10-way classification CNN into 10 binary ones without losing test accuracy [137]. However, sensAI is the first scheme to propose Class Parallelism and use it for low-latency, in-parallel model inference.

Network pruning: It is well-known in deep learning domain that traditional CNNs are over-parameterized [139][140]. To reduce computation cost and memory footprints, structured [141] and unstructured [142][143][144] network pruning gains lots of attention, which provide a effective way to reduce model size while maintaining decent accuracy [145][146].

sensAI also adopt network pruning methods. However, different from existing class-agnostic pruning methods, sensAI conducts one-shot, class-specific pruning to pull out binary or grouped classifiers from the base model. And we can further shrink down our subnets' sizes by incorporating traditional class-agnostic pruning methods [143][141][146].

One-Vs-All (OVA) reduction: OVA model reduction is a generic machine learning approach that reduces multi-class learning problem into a bunch of binary classification problems [147][148]. Previous literature demonstrates the effectiveness and theoretical correctness of OVA-based approaches [149][150]. Some modern art incorporates OVA with Error Correcting Output Codes (ECOC) to further increase the model serving accuracy [151][152][153]. However, both OVA approaches and its ECOC extensions need to *pre-define* the model structure of binary classifiers [154][151]. Different from OVA and its ECOC extensions, sensAI automatically learns varied subnet structures from the fully-trained base model for different classes or groups. And we achieve better model serving accuracy with less redundant binary or grouped subnets.

4.3 sensAI Method

In this section, we first provide an overview of sensAI workflow. Then we dive into all the key components in the following sections.

4.3.1 Overview

Before applying sensAI method, we assume to have a fully-trained N-way CNN classifier, where N is the total number of classes in the vision learning task. As a toy example depicted in Figure 4.4, sensAI has a 3-phase workflow: 1) class-specific pruning, 2) retraining, 3) combining results for in-parallel inference. Given a fully-trained base ConvNet, we first pull out binary or grouped classifiers from the base model. Second, we conduct retraining over these binary or grouped classifier to regain some possibly loss test accuracy due to the previous pruning step. Third, we deploy each subnet on a single GPU and conduct in-parallel, non-blocking model serving.

4.3.2 Class-specific Pruning

Here we first discuss how we distill binary classifiers from a base model. It is used for classification task with smaller number of classes. Then we extend our solution to distill grouped (i.e., multi-class) classifiers from the base model. This can be applied to more complicated classification problem with many classes.

4.3.2.1 Binary Classifiers

In order to pull out binary classifiers from a base model, we need to identify the sets of neurons that are crucial for particular image classes. We choose to use activation-based

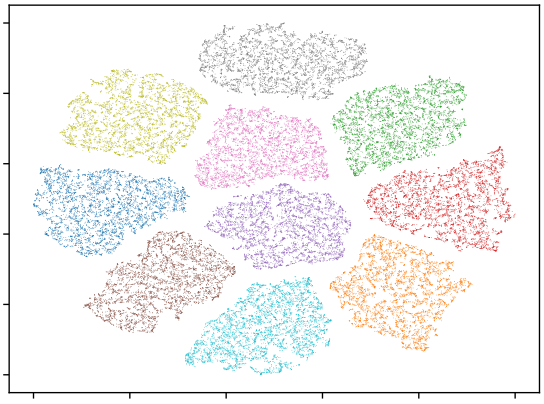


Figure 4.5: t-SNE visualization for feature representation of all training images with fully-trained VGG-19 on CIFAR-10 dataset.

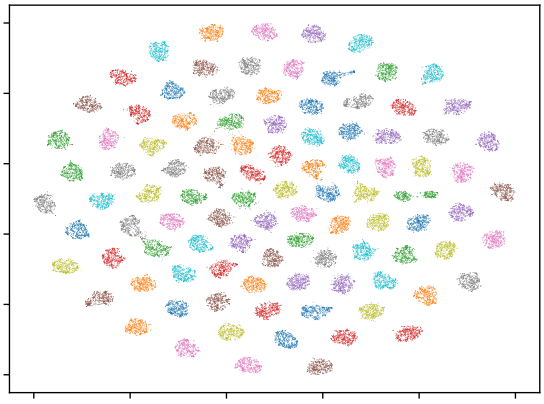


Figure 4.6: t-SNE visualization for feature representation of all training images with fully-trained VGG-19 on CIFAR-100 dataset.

pruning approach. Basically, given a fully trained ConvNet, we pass in all the training images of a single class into this ConvNet. Then we collects activation statistics per each neuron (i.e., channel). We directly borrow some popular activation-based pruning schemes such as Average Percentage of Zeros (APoZ [155]) and average activations (Avg [137]).

The pruning method is important and may affect our final in-parallel model serving accuracy [145]. After trails-and-errors test, we define our pruning policy as a hybrid of both Average Percentage of Zeros (APoZ [155]) and average activations (Avg [137]) with modifications on threshold values.

Our hybrid pruning policy is defined as below:

$$\Psi(N_{i,j}^{c}) = \frac{\sum^{D_c} \Phi(A_{i,j}^{c} < \theta_1)}{D_c} \tag{4.2}$$

$$where\ \Theta(A_{i,j}^{c}) < \theta_2 \tag{4.3}$$

In Equation 4.2, $N_{i,j}^{c}$ refers to j-th neuron in i-th layer that is evaluating on the c-th class of images, and the corresponding output feature-maps are $A_{i,j}^{c}$. The symbol D_c denotes the number of images that belonging to the c-th class. For $\Phi(\cdot)$, it calculates average percentage of activation values that are smaller than a threshold θ_1 for each image in the set of c-th class.

Our hybrid pruning policy works as follows. First, we generalize APoZ algorithm as $\Psi(\cdot)$ in Equation 4.2, which calcuate the average percent of activation values less than the

threshold value of θ_1. We generate our initialize pruning candidates' list by setting this predefined threshold θ_1. Second, we exclude the neurons from the pruning list if its feature-maps mean absolute value is larger than another threshold θ_2. By doing this two-step pruning, we can avoid pruning the neurons that generating featuremaps 1) with high percentage of near-zero value and 2) has a few very large non-zero values.

4.3.2.2 Grouped Classifiers

For simple classification tasks like CIFAR-10, it is reasonable to pull out 10 binary classifier from the base model. However, for more complicated tasks such as ImageNet-1K, it is unrealistic to distill 1000 binary classifiers from the base model and use 1000 GPUs together for in-parallel model inference.

In this case, given an N-way classification model, we decouple it into k m-way classifiers and ensure that $k \times m = N$. We call each m-way classifier as grouped model. To automatically group multiple classes, we adopt t-SNE and balanced k-mean clustering methods.

Our grouping methods works in two steps. First we evaluate the similarity of training images belonging to different classes. After passing in all the training images into a fully-trained CNN, we collects the feature vector before the last fully-connected layer as the feature representation for each training image. Then we use t-distributed stochastic neighbor embedding (t-SNE) [156] to cast the high-dimensional feature representation into 2D. Figure 4.5 and Figure 4.6 show the t-SNE vitualization results on both CIFAR-10 CIFAR-100 datasets. Here each data point with the same color refers to a training image belongs to the same class. The reason of using t-SNE is because it guarantees nearby classes are also closed with each other in the projected space [157]. Second, given the t-SNE results, we conduct balanced k-mean clustering [158][159][160] to automatically group nearby classes. Our balanced k-mean clustering is defined as Algorithm 1.

Algorithm 1 K-means clustering over t-SNE output

Input: total number of classes: N, number of groups: k, t-SNE output: $\{x_i^c\} \in X^c$ where c={0,1,..N-1}

for m = *0* ***to*** *N-1* **do**

  $a^m \leftarrow \sum_{n=0}^{|X^m|-1} x_n^m / \mid X^m \mid$; `// Avg FR per class`

end

Data: $\{a^c\} \in A^c$ for c={0,1,..N-1}, $Min_{size} \leftarrow \lfloor N/k \rfloor$, $Max_{size} \leftarrow \lceil N/k \rceil$

if $Min_{size} == Max_{size}$ **then**

  kmeans-balanced$(A^c, N/k)$

else

  kmeans-constrained$(A^c, Min_{size}, Max_{size})$

end

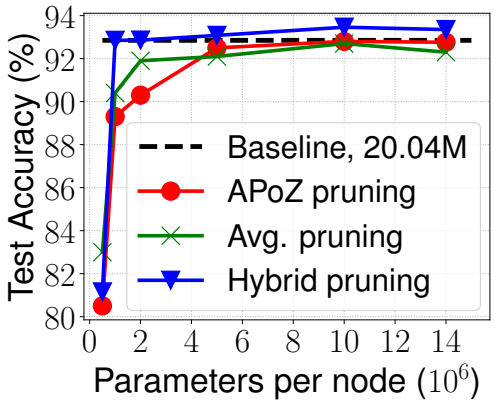


Figure 4.7: Pruning method comparison among APoZ, Avg, and our hybrid solutions (VGG-19, CIFAR-10).

In Algorithm 1, we first calculate all the average feature-representation (i.e., FR) per-class over t-SNE's output (i.e., X^c). Then we use these FR per-class as input data for k-means clustering algorithm [158].

The k-means clustering in Algorithm 1 mainly handles two cases. First, if the number of classes N could be divided by our group number as k, we directly apply *kmeans-balanced()* algorithm [159]. Second, if number of total class N is not divisible by number of groups k we have, we adopt another variation of balanced k-means called *kmeans-constrained()* [160] and pass-in minimum group size (i.e., Min_{size}) and maximum group size (i.e., Max_{size}) as arguments into the function. Here we split the classes as even as possible such that $Max_{size} - Min_{size} \leq 1$.

After we generates these class groups, we do group-wise pruning, which follows similar approach defined in Section 4.3.2.1. Instead of feed 1 class of training image, here we pass in a class group of training images into the fully trained base model. Then we can use the hybrid pruning policy to do group-wise pruning to distill grouped (i.e., multi-class) classifiers. For the final classification layer, we keep all m prediction heads and add another head to indicates the negative samples. Here negative samples refer to the input images that not belonging to this particular class group. Therefore, each of our grouped classifier have $m+1$ prediction heads.

4.3.3 Retraining

After getting all the binary or grouped classifiers from the base model, we need to conduct a retraining phase in order to regain possibly lost test accuracy because of previous pruning step. Note that all the binary or grouped classifier can be retrained independently, which means there is zero communication needed among the whole set of our binary or grouped classifiers.

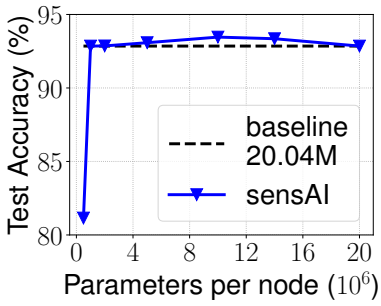


Figure 4.8: Number of Parameters vs test accuracy comparison (VGG-19, CIFAR-10).

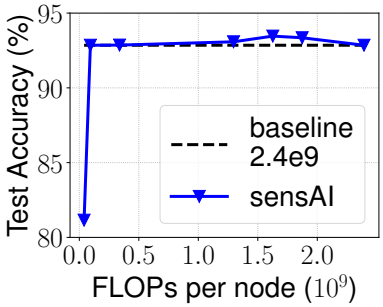


Figure 4.9: FLOPs consumption vs test accuracy comparison (VGG-19, CIFAR-10).

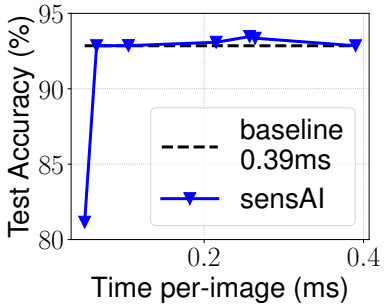


Figure 4.10: Per-image inference time vs test accuracy comparison (VGG-19, CIFAR-10).

To improve retrain efficiency, we need to balance both negative and positive training samples. We also need to modify the loss function accordingly. For binary classifiers, we need to change the loss function into binary cross-entropy (BCE) loss. And for grouped classifiers, we need to change it into multi-way cross-entropy loss with $(m + 1)$ prediction heads.

4.3.4 Combining Results Back to N-way Predictions

After getting all our retrained binary or grouped classifiers, we combine their outputs to the original N-way prediction tasks. Here we directly pick the maximum probability of being positive among all our binary/grouped classifiers' output as the final prediction result of original N-way prediction tasks. For example, in Figure 4.4, since cat confidence value 0.6 is larger than dog's confidence value 0.2, we predict the image as a cat image.

4.4 Evaluation

4.4.1 Datasets and Models

We evaluate sensAI on several standard CNN models, such as ResNet [6] and VGGNet [102]. We test our approach on three different datasets as CIFAR-10, CIFAR-100 [14] and ImageNet-1K [15]. In addition, we also verify sensAI's effectiveness over some more efficient networks such as ShuffleNet-V2 [161] and MobileNet-V2 [162]. We report our model size and FLOPs reduction, inference time speedups and Top-1 test accuracy performance.

For the hardware testbed, we use Nvidia Tesla M60 GPUs [163] for all the experiments. For each binary or grouped classifier, we assign it to a single GPU exclusive without resource sharing among other workloads.

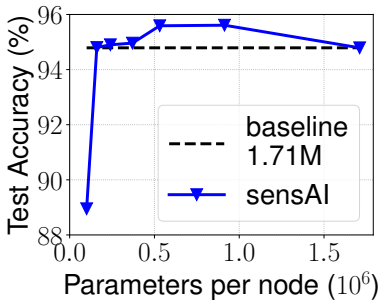


Figure 4.11: Number of Parameters vs test accuracy comparison (ResNet-164, CIFAR-10).

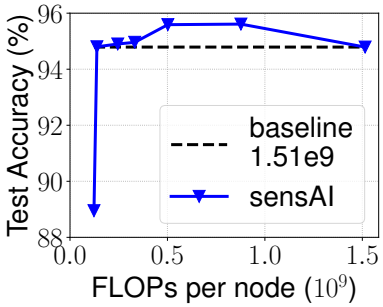


Figure 4.12: FLOPs consumption vs test accuracy comparison (ResNet-164, CIFAR-10).

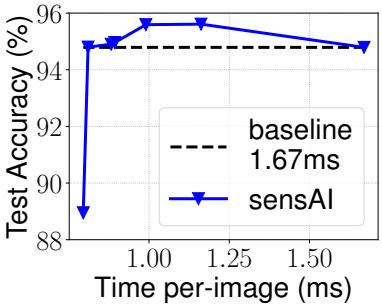


Figure 4.13: Per-image inference time vs test accuracy comparison (ResNet-164, CIFAR-10).

4.4.2 CIFAR-10 Results

Here we mainly evaluate on two CNNs as VGG-19 and ResNet-164. We also test our improvement over more efficient CNNs like MobileNet-V2 and ShuffleNet-V2. We train the model with standard hyper-parameter settings. We train 164 epochs. Learning rate starts with 0.1 then decay by 0.1 at 91, 122 epochs. The test accuracy of our base models are 92.85% for VGG-19, 94.79% for ResNet-164, 93.46% for ShuffleNet-V2 and 94.24% for MobileNet-V2.

4.4.2.1 Pruning Policy Comparison

We first compare different pruning policies we have, namely APoZ, avg, and our hybrid policy. We compare the final test accuracy using these three policy on VGG-19 using CIFAR-10 dataset.

For our hybrid policy defined in Section 4.3.2.1's equation 4.2 and Equation 4.3, we need to find two threshold values as θ_1 and θ_2. In our implementation, these two threshold are determined by a grid search on separate validation dataset. And the thresholds values are not layer-specific but global across different layers of the neural nets. The pruning percentage is thus determined by these two threshold values.

As depicted in Figure 4.7, our hybrid pruning policy generally outperforms over both APoZ and Avg policies. It achieves the highest inference accuracy among the three policies for the binary classifiers of same size. Therefore, we adopt our hybrid pruning policy as the default pruning scheme for the rest evaluations.

4.4.2.2 sensAI Evaluation on VGG-19 and ResNet-164

We evaluate sensAI performance with respects of number of parameters per GPU, FLOPs consumption, and inference time per-image. We mainly test on VGG-19 (with batch normal-

	Model	Test acc. (%)	Avg. per-GPU model size (10^6)	Per-image infer. time (ms)
MP-baseline (10 GPUs)	VGG-19	92.85	2.00	1.12
	ResNet-164	94.79	0.17	2.33
sensAI	VGG-19	92.86	1.01	0.06
	ResNet-164	94.79	0.16	0.81

Table 4.1: Comparison between baseline with model parallelism (MP) and sensAI using 10 GPUs.

ization [58]) and ResNet-164. For the retraining of our binary classifiers, we borrow similar learning rate decay policy as described in Section 4.4.2.

Figure 4.8, Figure 4.9 and Figure 4.10 show the evaluation results of VGG-19 on CIFAR-10 dataset. Surprisingly, by only single-shot class-specific pruning, we can reduce the model size by 20x per GPU (Figure 4.8), FLOPs by 24x with no test accuracy loss (Figure 4.9). For per-image inference time, we can reduce it by 6x as depicted in Figure 4.10.

Similar results can be found in ResNet-164 evaluations as Figure 4.11, Figure 4.12 and Figure 4.13. Comparing with base model of ResNet-164, sensAI can reduce number of parameter by 11x (Figure 4.11), FLOPs consumption by 11x (Figure 4.12) and inference time per image by 2x (Figure 4.13) without any test accuracy loss. The inference time reduction in ResNet-164 is worse compared with our VGG-19 results. The main reason is we only reduce the width of the model and keep the model depth unchanged. It is generally believed model pruning should not prune in the model depth dimension [141][142]. Therefore, 2x per-image inference time speedup on ResNet-164 is still good.

Why we can prune so many neurons with just single-shot pruning? The main reason is we simplify the classification task itself. More specifically, in CIFAR-10 case, we simplify the original 10-way prediction problem into a few binary prediction problems. Thus, the base model has more redundancy for us to achieve this high ratio of pruning. In addition, we can incorporate class-agnostic pruning methods [143][141] to further reduce the size of our binary models.

Another thing worth mentioning is that, we can achieve better test accuracy when maintaining slightly large binary models. For instance, in ResNet-164 results shown as Figure 4.11, better test accuracy can be achieved when the average binary model size is ranging from 0.16M to 0.91M.

4.4.2.3 sensAI vs Model Parallel Baseline

Before we compare sensAI with base model assigned to single GPU, we now evaluate sensAI with model parallel inference, where both sensAI and model parallelism using 10 GPUs.

	Test Accuracy %	Average binary model size (10^6)
OVA	91.65	0.27
sensAI	94.79	0.16
	94.90	0.24

Table 4.2: Comparison between OVA and sensAI with 10 GPUs on CIFAR-10.

For model parallel baseline, as suggested by recent model parallel work [35][36], we evenly split the model layers across all the 10 GPUs. The follow model parallelism baselines in our CIFAR-100 and ImageNet-1K follows the same role.

In Table 4.1, adopting model parallel inference performs worse than doing single image inference on a single GPU. More specifically, comparing with single-GPU baseline, our model parallel baseline's per-image inference time is 0.73ms longer on VGG-19, and 0.66ms longer on ResNet-164. The main reasons are twofold. First, there exists high communication overhead among all the GPUs in use during the model parallel inference. Second, model parallelism naturally creates synchronization barries due the the sequential dependency of DNN layers [85]. For example, the GPUs holding last few layers cannot start working until fully receiving activations from all the previous layers.

In contrast, the binary classifiers generated by sensAI can run independently without blocking each other. Thus, in Table 4.1 compared with model parallel baseline, for single image inference time, sensAI can achieve 18x speedup in VGG-19 and 3x in ResNet-164 without losing test accuracy.

4.4.2.4 sensAI vs OVA

One-Vs-All (OVA) reduction scheme is studied for multiple years [147][148][149], which is close to our sensAI approach. Basically, for image classification tasks, both sensAI and OVA decouple a multi-way classification problem into a bunch of much simpler, binary classification problems.

Here we direction compare OVA performance with sensAI on CIFAR-10 dataset. Since OVA needs pre-defined model structure for binary classification tasks, we directly choose ResNet-20 for each single class task in CIFAR-10 dataset. We follow the same training recipes described in Section 4.4.2. For training datasets of OVA's binary classifiers, we use the same per-class dataset we generated at our sensAI's retraining stage. After all the OVA's binary classifiers are fully trained, we also impose calibrations among all the OVA's binary classifier in order to obtain the possible best test accuracy.

For evaluation, we use these calibrated and fully trained OVA binary classifier together for in-parallel model serving as our sensAI approach. For our sensAI models, we directly use two sets of binary models pulled from ResNet-164 in Section 4.4.2.2. One set of models are

	Model	Test acc. (%)	Model size per-GPU (10^6)	FLOPs per-GPU (10^9)	Per-image infer. time (ms)
Baseline	MobileNet-V2	94.24	2.30	0.56	0.48
	ShuffleNet-V2	93.46	1.26	0.27	0.23
sensAI	MobileNet-V2	94.27	0.45	0.16	0.22
	ShuffleNet-V2	93.50	0.31	0.07	0.12

Table 4.3: Comparison between efficient baseline models and sensAI.

smaller than OVA's ResNet-20. The other set of binary models share similar model sizes as OVA solution.

As shown in Table 4.2, when using similar model sizes as OVA's average size of 0.27M and sensAI's are with size of 0.24M, our binary classifiers outperforms OVA solutions with 3.25% high test accuracy. For the case that sensAI's binary models are much smaller than OVA's ResNet-20 (i.e., sensAI average model size is 0.16), our sensAI solution still achieve higher model serving accuracy and outperforms over OVA counterpart by around 3% on CIFAR-10 dataset.

In summary, compared with OVA solution of same and pre-defined model structure, sensAI automatically learns different model architecture for different class of images from the base model. And our solution achieves better model serving accuracy while having less computation cost.

4.4.2.5 sensAI Improvements on Efficient CNNs

Now we verify the effectiveness of our sensAI approach on more efficient ConvNets such as MobileNet-V2 [162] and ShuffleNet-V2 [161].

These two models are designed with the goal of computational efficiency. Both MobileNet-V2 and ShuffleNet-V2 are small regarding to number of model paramerters and FLOPs consumptions. And they are designed to be executed on single GPU. In addition, as results shown in Section 4.4.2.3, small model generally performs worse with model parallelism. Therefore, here we report sensAI's speed-up over single GPU baselines.

As results shown in Table 4.3, for MobileNet-V2, sensAI can reduce number of parameters by 5x, with 3.5x FLOPs reduction and 2x per-image inference time reduction. For ShuffleNet-V2, sensAI reduces number of parameters by 4x, with 4x FLOPs reduction, and 2x per-image inference time reduction.

Our sensAI method is generic approach which can also be directly applicable to modern ConvNets [164] like ConvNeXt [165], or even with 3D input data [166].

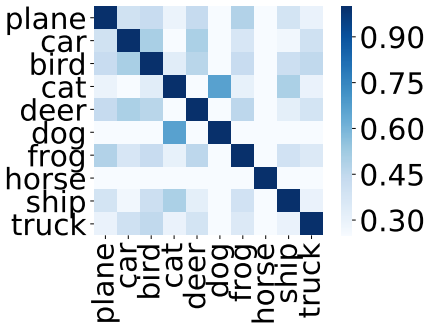


Figure 4.14: Similarity comparison among binary classifiers by measuring IoU on channels (VGG-19, CIFAR-10).

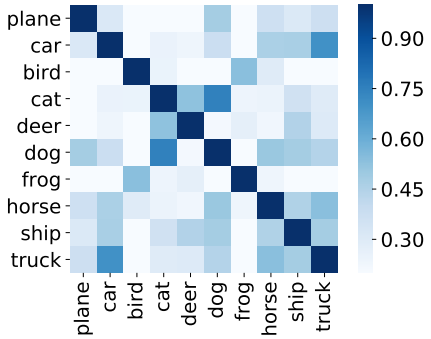


Figure 4.15: Similarity comparison among binary classifiers by measuring IoU on channels (ResNet-164, CIFAR-10).

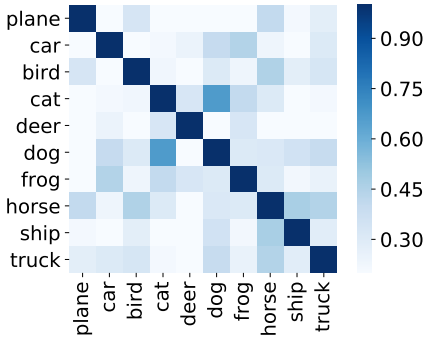


Figure 4.16: Similarity comparison among binary classifiers by measuring IoU on channels (ShuffleNet-V2, CIFAR-10).

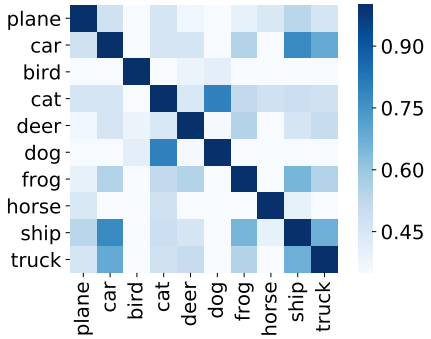


Figure 4.17: Similarity comparison among binary classifiers by measuring IoU on channels (MobileNet-V2, CIFAR-10).

4.4.2.6 Binary Models Analysis

Here we do simple statistical analysis over the binary models we distilled from the base models in CIFAR-10 dataset. The sets of binary models we analyzed are the best ones we have. Here best means the binary models with smallest number of parameters while maintaining the same test accuracy as the base model. We analyze the best binery model sets we pulled out from different base models, namely VGG-19, ResNet-164, ShuffleNet-V2 and MobileNet-V2.

We first analyze the similarity among the set of binary classifiers we collected. For each binary classifier we pulled out from the same base model, we collect its channel indices in the original base model. Then we measure the similarity of two binary classifier by taking the

Type	Model	Grouping method	Test acc. (%)	Model size (10^6)	FLOPs (10^9)	Per-image infer. time speed-up (over 1-GPU Baseline)
Baseline (1 GPU)	VGG-19	N/A	71.95	20.02	2.39	1x
	ResNet-110	N/A	72.44	1.66	1.3	1x
MP-baseline (5 GPUs)	VGG-19	N/A	71.95	4.00	0.48	0.41x
	ResNet-110	N/A	72.44	0.33	0.29	0.78x
MP-baseline (10 GPUs)	VGG-19	N/A	71.95	2.00	0.24	0.33x
	ResNet-110	N/A	72.44	0.17	0.14	0.76x
sensAI	VGG-19	Random 5-group	71.75	4.28	1.06	2.17x
		Random 10-group	71.84	2.21	0.36	3.25x
		Nearby 5-group	72.07	3.01	0.47	2.44x
		Nearby 10-group	72.18	1.61	0.21	4.5x
	ResNet-110	Random 5-group	72.43	0.513	0.501	1.64x
		Random 10-group	72.14	0.24	0.233	1.87x
		Nearby 5-group	72.52	0.31	0.298	1.8x
		Nearby 10-group	72.41	0.167	0.143	2.06x

Table 4.4: Comparison between sensAI with two grouping methods (random and nearby grouping) with 5-group (5 GPUs) 10-group (10 GPUs) v.s. baselines of Single GPU and model parallelism (MP) using same amount of GPUs (5, 10 GPUs) on CIFAR-100.

intersection of unions (IoU) of the channel indices. Figure 4.14 shows the similarity of binary classifiers we collected from VGG-19 as a heatmap. In this heatmap, the darker color block indicates the the binary pair has higher similarity. We observe that cat binary classifier and dog binary classifier share most neurons among all pairs, which is very intuitive since they look similar in human eyes. Similar observation results are found in other popular neural nets such as ResNet-164, ShuffleNet-V2 and MobileNet-V2, which are shown as Figure 4.15, Figure 4.16 and Figure 4.17 respectively.

Second, we analyse the model sizes amoung 10 binary classifiers distilled from the same base model. We observe that the whole set of binary classifiers usualy share the same model sizes. For example, in VGG-19, all binary models have around 1 million parameters with variance less than 0.1M. In ResNet-164, binary classifiers all have around 0.16M parameters with variance less than 0.02M. For the case of more efficient ConvNet as MobileNet-V2, the binary ones are all 0.45M parameters with less than 0.05M variance. For ShuffleNet-V2, all binary models are around 0.31M parameter with less than 0.04M variance. The reason behind may due to the fact that we split the original task evenly among subnets. This decent attribute guarantee all our binary classifiers can finish their in-parallel inference at roughly the same time, which means there is no stragglers among the whole set of binary classifiers in use.

4.4.3 CIFAR-100 Results

Starting in this section, we switch our evaluation from binary classifiers to the grouped classifiers we generate using sensAI approach. Here we present our results on CIFAR-100

dataset. We first train our base models as VGG-19 and ResNet-110. With similar training recipe as mentioned in Section 4.4.2, we got test accuracy as 71.95% for VGG-19 and 72.44% for ResNet-110 on CIFAR-100 dataset. These two models are served as the baseline for both single GPU (i.e., Baseline in Table 4.4) and model parallelism using multiple GPUs (i.e., MP-baseline in Table 4.4) cases. Similar as described in Section 4.4.2.3, for model parallel baseline on CIFAR-100, we also evenly split these ConvNets model layers across all the GPUs in use. For automatic grouping, we mainly compare two grouping methods, namely random grouping and nearby grouping using t-SNE and k-means algorithm as we defined in Section 4.3.2.2.

As discussed in Section 4.4.2.6, by evenly split the original tasks, we can get subnets with similar model sizes, which means there is no straggler during in-parallel model serving using the whole set of subnets. Following similar rule and achieving good load-balancing, we assign each of our grouped classifier with same number of classes coverage. We tested two group size settings, 5 and 10 groups. In 5-group case, each grouped classifier covers model serving of 20 classes. In 10-group setting, each grouped classifier is responsible for 10-class prediction.

As we discussed in Section 4.4.2.1, we choose hybrid pruning policy by default in our evaluation here on CIFAR-10 dataset, since it provides the highest model serving accuracy given the same model pruning ratio. We also adopt this hybrid pruning policy for our later evaluation on ImageNet-1K dataset in Section 4.4.4. For retraining process over our grouped classifiers on CIFAR-100 dataset, since each class only has 500 training images, it is more likely to cause over-fitting. Thus, we reduce the number of retraining epoch to only 40, and collects the best set of grouped classifiers we have during this training session.

4.4.3.1 sensAI vs single GPU Baseline

In Table 4.4, without sacrificing too much test accuracy, we collect the set of grouped models with smallest size. In general, comparing with random grouping, our nearby grouping described in Section 4.3.2.2 performs better in both model serving accuracy and per-image serving time reduction.

Random Grouping: As shown in Table 4.4, for 5-group cases, given test accuracy drop within 0.3% of baseline, we can reduce model size by 4.6x/3.2x, FLOPs consumption of 2.2x/2.6x and per-image inference time by 2.17x/1.64x over single GPU baseline (i.e., Baseline in Table 4.4) on VGG-19/ResNet-110, respectively. In the 10-group cases, with negligible test accuracy loss, we achieve model size reduction by 9x/7x, FLOPs reduction by 6x/5x, and 3.25x/1.87x inference time reduction of single image on VGG-19/ResNet-110, separately.

Nearby Grouping: In Table 4.4, our nearby grouping method generally outperforms the random grouping counterpart. With 5 groups, when comparing with single-GPU baseline, we achieve model size reduction by 6.6x/5.3x, FLOPs reduction by 5x/4x, and per-image serving time reduction by 2.44/1.8x on VGG-19/ResNet-110 separately. In our 10-group settings, with negligible inference accuracy loss, our nearby grouping method can achieve

Type	Model	Nearby Grouping	Test acc. (%)	Model size (10^6)	FLOPs (10^9)	Per-image infer. time speed-up (over 1-GPU Baseline)
Baseline (1 GPU)	VGG-19	N/A	74.21	143.68	117.99	1x
	ResNet-50	N/A	76.13	25.56	24.63	1x
MP-baseline (10 GPUs)	VGG-19	N/A	74.21	14.37	11.8	0.45x
	ResNet-50	N/A	76.13	2.56	2.47	0.58x
MP-baseline (20 GPUs)	VGG-19	N/A	74.21	7.2	5.9	0.43x
	ResNet-50	N/A	76.13	1.28	1.24	0.57x
sensAI	VGG-19	10-group	74.26	15.34	18.45	3.4x
		20-group	74.24	8.57	9.73	5.87x
	ResNet-50	10-group	76.17	2.71	2.93	2.16x
		20-group	76.06	1.83	1.89	3.08x

Table 4.5: Comparison between sensAI of 10-group (10 GPUs) and 20-group (20 GPUs) v.s. baselines of Single GPU and model parallelism (MP) with 10 GPUs and 20 GPUs on ImageNet-1K.

reduction of 12x/10x on model size, 11x/9x on FLOPs consumption and 4.5x/2.6x per-image serving time on VGG-19/ResNet-110, respectively.

4.4.3.2 sensAI vs Model Parallel Baseline

Here we compare the grouped classifiers from sensAI against model parallel baseline (i.e., MP-baseline in Table 4.4) using the same number of GPUs. we test in two different cases, 5-GPU case and 10-GPU case. As mentioned in Section 4.4.2, for model parallel baseline, we evenly split model layers among all the GPUs in use.

As shown in Table 4.4, for model-parallel baseline, even though the per-GPU model sizes and FLOPs are significantly reduced, it generally performs much worse compared with single-GPU baseline with respect to per-image serving time. The key reason is model partitions hold on different GPUs may block each other due to sequential dependency of the DNN layers [85]. We present our results on both 5-GPU and 10-GPU cases.

5-GPU: Comparing with 5-GPU model parallel baseline, sensAI can maintain similar model size reduction and FLOPs reduction for both VGG-19 and ResNet-110. However, since all our 20-class grouped models can run in-parallel without blocking each other, we can achieve 2.44/0.41=5.95x and 1.8/0.78=2.3x speedup over model parallel baseline for per-image model serving on VGG-19/ResNet-110, separately.

10-GPU: Similar results can also be found in this 10-GPU setting. Comparing with 10-GPU model parallel baseline, our 10-class grouped classifiers achieves slightly higher reduction rate in terms of model size and FLOPs consumption per-GPU for both VGG-19 and ResNet-110. In single image inference time, sensAI achieves 4.5/0.33=13.64x and 2.06/0.76=2.71x time reduction over model-parallel counterparts on VGG-19 and ResNet-110, separately.

4.4.4 ImageNet-1K Results

In this section we present our experimental results on ImageNet-1K dataset [15]. The two model we evluate here are VGG-19 and ResNet-50. Here we directly use pre-trained models as our baselines (i.e., Baseline in Table 4.5).

In our sensAI setup, we use the same hybrid pruning policy, due to its advantages over APoZ and Avg policies discussed in Section 4.4.2.1. As evaluated in Section 4.4.3.1, since nearby grouping generally outperforms random grouping method, we apply nearby grouping strategy and form 10-group and 20-group settings on ImageNet-1K dataset. For 10-group case, each grouped classifier is for predicting 100 classes. In 20-group setting, each grouped classifier is responsible for decision making of 50 classes. During our retraining stage, we limit the number of epochs to be 40. For the retraining recipe, we have learning rate decay of 0.1 at both 20th and 30th epochs. We report our best sets of grouped classifiers (i.e., smallest model size with negligible test accuracy loss) as Table 4.5.

4.4.4.1 sensAI vs Single GPU Baseline

In Table 4.5, by splitting VGG-19 into 10 grouped classifiers using sensAI, we reduce the model size by 9x, FLOPs by 6x and per-image inference time by 3.4x over single GPU baseline at no loss of model serving accuracy. In the 20-group setting, we achieve model size reduction by around 17x, FLOPs reduction by 12x and reduction of inference time per-image by around 6x (i.e., 5.87x in Table 4.5) without test accuracy loss.

For ResNet-50 results shown in Table 4.5, by decoupling base model into 10 grouped classifiers using sensAI, we can reduce each grouped model size by 9.4x, FLOPs consumption by 8.4x and per-image serving time by 3.08x over single GPU baseline with moderate test accuracy loss (i.e., 0.07%). Comparing with deeper ResNet (i.e., ResNet-164 and ResNet-110) we used in our CIFAR-10 and CIFAR-100 experiments, here with shallower ResNet like ResNet-50, we can indeed prune more neurons (i.e., channels) and achieve higher speedups for per-image serving time.

One thing worth noting on ImageNet-1K evaluation is, since image resolution of each ImageNet-1K picture is much larger than CIFAR's 32x32 pixels, sensAI can achieve per-image inference time reduction at millisecond level. For instance, in our 20-group with VGG-19 setting, we can reduce 7.79ms of per-image serving latency over single GPU baseline with no test accuracy loss.

Another thing worth mentioning is that, each of our grouped classifier can be retrained independently, since there is zero communication needed at retraining stage among the whole set of our grouped classifiers. Given the longer training time on ImageNet-1K dataset comparing with CIFAR-10 and CIFAR-100, we can incorporate data parallelism to further accelerate our model retraining stage. More specifically, for each of our grouped classifier, we adopt data parallel training to increase parallelism and training throughput. Thus, we can make trade-off between training completion time and number of GPUs in use.

We note the aggregated number of parameters among the whole set of our grouped classifiers is larger than the base model. Given that our main goal is to reduce single image serving latency not total number of parameters, our class parallelism is still a practical and generic solution. Since we only do single-shot class-specific pruning, we can further incorporate class-agnostic pruning methods [141][143] to further reduce our grouped model sizes.

4.4.4.2 sensAI vs Model Parallel Baseline

Now we evaluate our sensAI approach and compare it with model parallel baseline on ImageNet-1K dataset. For model parallel baseline, we evenly split base model over the same amount of GPUs that our sensAI uses. In Table 4.5, we report the comparison results between our method and model parallel baseline (i.e., MP-Baseline (10 GPUs) and MP-Baseline (20 GPUs) in Table 4.5).

10-GPU: We first present our result in 10-GPU case. As shown in Table 4.5, comparing with 10-GPU MP-Baseline, 10-group classifiers in sensAI maintains larger model size and FLOP consumption per-GPU. However, regarding per-image model serving time, sensAI achieves 3.4/0.45=7.5x and 2.16/0.58=3.7x end-to-end latency reduction on VGG-19 and ResNet-50, respectively.

20-GPU: For 20-GPU setting, compared with the corresponding model parallel baseline (i.e., MP-Baseline (20-GPU) in Table 4.5), the 20-group classifiers from sensAI achieve time reduction for single image by 5.87/0.43=13.5x and 3.08/0.57=5.4x on VGG-19 and ResNet-50, separately.

4.5 Discussion

Above we describe sensAI's advantages over traditional data and model parallel model serving. Here we discuss several issues we note on sensAI's novel class parallelism approach.

First, in our defined class parallelism paradigm, we have a hard limitation: we can only scale out up to the number of classes we have in the original classification task. This could limit its usage in some cases. For instance, we cannot distribute a 10-way classification task (e.g., CIFAR-10) to over more than 10 GPUs or nodes. Furthermore, if the original problem is a binary classification task itself, our methodology is not directly applicable.

Second, we note that the aggregated number parameters among the whole set of our binary/grouped classifiers may even be larger than the base model itself (e.g., ImageNet-1K results in Section 4.4.4.1). Since our main goal is to reduce single image serving latency, reducing total amount of parameters can be regarded as a byproduct but not necessary. To further reduce our live data serving latency and mode size, besides our single-shot class-specific pruning, we can incorporate class-agnostic pruning methods [141][142][167][168][169][170][171] for iterative pruning, and quantization schemes [143][172][173][174][175][176] for model compression. And we leave it as a future research direction.

Third, if the set of our binary/grouped classifiers have high variance in terms of number of parameters, the end-to-end in-parallel inference time is then determined by the ones with most computation. Given our analysis on binary classifiers in Section 4.4.2.6, we find that all our binary classifiers share similar model sizes. This attribute of similar subnet sizes may because of our even split of original classification task. This feature of similar model sizes guarantees that all our binary/grouped classifiers can finish in-parallel forward propagation at roughly the same time (i.e., no stragglers).

Fourth, with respect to baseline, single GPU setting achieves the lowest latency for single image inference in most cases. This is mainly because transferring intermediate results (e.g., feature-maps) across different GPUs and machines can be expensive. And conventional model parallelism naturally creates hard synchronization barriers in our single image inference scenarios. For example, the GPUs holding deeper layers of the model cannot start forward propagation until fully receiving activations from GPUs maintaining previous model layers. Furthermore, neither data parallelism [21] nor pipeline parallelism [35] is applicable for reducing serving latency in single image cases. Our evaluation results also verify that, even though model parallel baseline reduces per-GPU model sizes and FLOPs, the end-to-end per-image serving latency is usually higher than single GPU baseline. In contrast, sensAI achieves similar per-GPU model sizes as model parallel baseline while reducing model serving latency over single-GPU baseline at the same time. The key reason is we remove all possible communication overheads and synchronization barriers in forward propagation. Thus, all our subnets are able to run in-parallel without blocking each other.

Fifth, besides class parallel model serving, we further extend our methodology to class-specific, concurrent model training at zero communication cost. We briefly discuss the method and evaluate it on CIFAR-10 dataset as Section 4.6.

4.6 Extending to Model Training

We describe how we extend our class parallelism to model training stage at no communication cost.

4.6.1 Method

Applying class parallelism at training stage is slightly different from the inference stage we discussed above. Given that we do not have a pre-trained base model, we first initialize the model parameter and train a base model for several epochs, which allow the model to learn feature representations from the training data samples. Then, based on this "half-baked" base model, we distill our binary or grouped classifiers with same methodology we described in Section 4.3.

Given this "half-baked" N-way classification model, we first collect the activation statistics of each neuron (i.e., channel) by feeding in all the training images for each class/class-group. Then we use our hybrid pruning policy to pull out binary/grouped classifiers. Second,

we deploy each of our binary/grouped classifier on a single GPU for independent class-parallel training. After all the subnets are fully trained, we use them together for in-parallel model serving.

In class-parallel model training, even though we need to train the base model for a small number of epochs, most of the training epochs are conducted over our whole set of binary/grouped classifiers at zero communication cost. For example, on CIFAR-10 dataset, it normally takes around 164 epochs to fully train a base model [177]. For our class-parallel model training, we empirically train the base model for only 20 epochs to get our "half-baked" baseline, from which we distill our binary classifiers. Then the rest 164-20=144 epochs are trained on the binary classifiers in-parallel at no communication cost. Since each of our binary classifier is much smaller in size when comparing with base model and can be trained in-parallel over multiple GPUs, the total training completion time of class parallelism is much shorter than training base model on single GPU. After all our binary/grouped classifiers are fully trained, we combine their output values for the original N-way predictions.

4.6.2 Results

We evaluate class-parallel training with VGG-19 and ResNet-164 on CIFAR-10 dataset.

As mentioned in Section 4.6.1, we first initialize weights for the base model and train it on full dataset for 20 epochs. Given that the model converges fastest during the first 10s of epochs, we can get a "half-baked" model which has high similarity compared with a fully trained one. We conduct our one-shot class-specific pruning to pull out binary classifiers from this "half-baked" model. Then we form per-class training dataset and train each of these binary classifiers for the rest 164-20=144 epochs. Here we use 64 as the mini-batch size during training. We measure per-iteration training time both both base model and our in-parallel binary ones. After all the binary classifiers are fully trained, we use them together for serving and report our model size and training time reduction.

Without losing test accuracy, we get binary models with around 1M parameters for VGG-19 and 0.16M for ResNet-164. For class-parallel model training, the completion time reduction is 11x on VGG-19, which is over linear scalability. The key reason is we reduce model size per-GPU by 20x and zero communication is needed during our in-parallel model training. For ResNet-164, we can reduce the training completion time by 3x.

We also conduct binary model analysis for these binary classifiers distilled from "half-baked" base model. Generally speaking, compared with binary models pulled from fully trained base model (Section 4.4.2.6), these binary subnets getting from "half-baked" baseline has lower top-1 test accuracy with drop between 0.04% to 0.38%. In addition, these binary models generated from "half-baked" base model have higher variance in model size. For instance, some of binary models can be 1.2x to 2x the size of some other smaller ones in the same set. This is possibly because that the baseline model is not fully trained, which generates more noisy signals at our class-specific pruning stage. However, we still maintain 2x to 5x speedups for per-image inference over model parallel counterparts. The main reason is

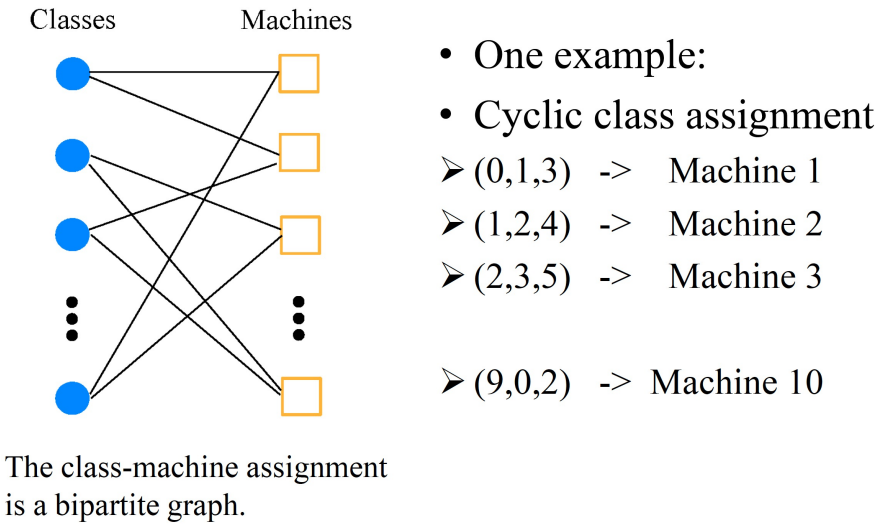


Figure 4.18: Robust Class Parallelism with Cyclic Coding.

our binary models are very tiny and double the size does not make much difference regarding to the end-to-end forward propagation time during live data serving.

4.7 Extending to Fault Tolerance, Robotic Control, and Beyond

To improve system robustness, we add fault tolerance feature on top of our vanilla class parallelism approach. We borrow simple error-correcting codes [178][179] called cyclic coding [180] for our fault tolerance extension.

As shown in Figure 4.18, different from our vanilla class parallelism where grouped classifiers having disjoint class coverage, we now allow all grouped classifiers have overlapped class coverage, which is similar as parity models [179]. Thus, for each overlapped class, we can do major vote over multiple prediction values of the same class to avoid the cases like machine failures or malicious nodes. For example, in Figure 4.18, class #2 is covered among machine 2 (i.e., (_,2,_)), machine 3 (i.e., (2,_,_)) and machine 10 (i.e., (_,_,2)). Thus, For class #2, we can do major vote over the three confidence values generated from machine 2,3 and 10. More details and preliminary results on CIFAR-10 and CIFAR-100 datasets can be found in our robust sensAI paper [181].

We also extend sensAI's divide-and-conquer scheme to control system in drones. Right now the control system on a quad-copter is centralized. The centralized control module collects data from the sensors located around four propellers, and then send back decisions to all 4 propellers. In our sensAI reinforcement learning branch (i.e., sensAI-RL), we want to decouple the centralized control system into four much smaller control units, where each

is only responsible for decision making of a single propeller. More details can be found in our sensAI-RL paper [182].

Since sensAI's class parallelism is a generic approach, we are also working on extending it to other deep learning models such as recurrent neural networks [183][184], transformers [8][13][146], etc.

4.8 Summary

In sensAI, we propose a novel in-parallel model serving paradigm called class parallelism, which reduces serving latency on live data. Given a fully trained base model, we decouple it into disconnected subnets, where each is responsible for decision making of a single class/class-group. Our experiments on CIFAR-10 demonstrate its effectiveness: our class parallelism can reduce per-image serving time by 6x on VGG-19 and 2x across ResNet-164, MobileNet-V2, ShuffleNet-V2 over the best baseline among single-GPU and model-parallel settings. Comparing with model parallel baseline, we can reduce upto 18x model serving latency on single image. Our evaluations on CIFAR-100 and ImageNet-1K show similar results, which also verifies sensAI's effectiveness. We further extend our class parallelism to the model training stage. We then add fault tolerance feature to sensAI's class parallelism with cyclic coding. We also study the possibility for extending class parallelism to reinforcement learning, natural language processing areas.

Chapter 5

Improving on-device memory utilization

5.1 Background

Given giant DNNs and huge amount of input data, distributed model training using multiple accelerators has been widely-adopted. To boost up model training speed, data parallelism [22] and model parallelism [20][82] are two most popular approaches. To increase parallelism, these schemes either partition input data or split model across multiple accelerators.

In distributed DNN training, all the training tasks are required to be launched at the same time, i.e., gang-scheduled [49][51][185][186][187][188]. It is mainly because of two reasons. First, some hyper-parameters (e.g., learning rate, mini-batch size, etc.) need to be picked with fixed number of machines or accelerators [21][189][190]. Second, synchronization occurs frequently among all the accelerators of the same job [50]. Thus, it imposes a strong constraint that all the in-parallel model training tasks should start and end at the same time to avoid stragglers [191].

Despite of the immense popularity of gang-scheduling policy in distributed DNN training phase, it may under-utilize memory and computation resources on the accelerators. From on-device memory aspect, gang-scheduling policy forces all the GPUs to reach their peak and valley memory usage at the same time. Thus, in single job case, the memory valley periods are wasted among all the GPUs. On the computation side, for most advanced Nvidia GPUs like H100 [101], A100 [100] and V100 [96], the computation power of single GPU is increasing tremendously. However, the computation resources are often under-utilized due to limited on-device memory size. A wide range of deep learning jobs are memory bounded, and computation cores are insufficiently used [39]. In Figure 5.1 and Figure 5.2, we monitor resource utilization of gang scheduled data parallel training job using 2 V100 GPUs [6]. We show normalized GPU memory usage (Figure 5.1) and computation core usage (Figure 5.2). In Figure 5.1, on-device memory resource is repeatedly underutilized during

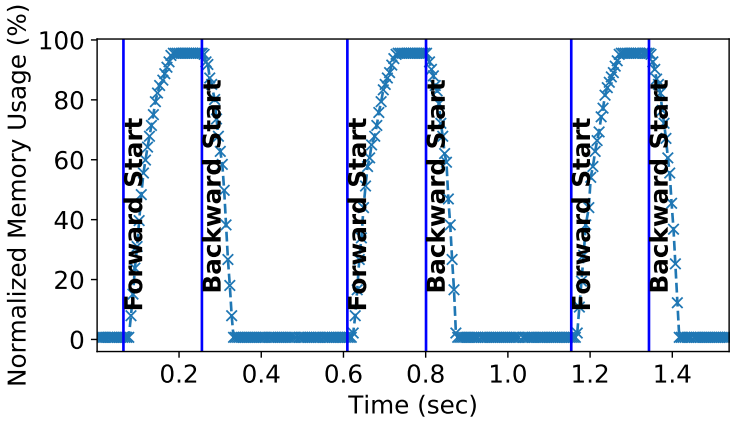


Figure 5.1: Normalized on-device memory usage of data parallel training job using 2 V100 GPUs with gang-scheduling.

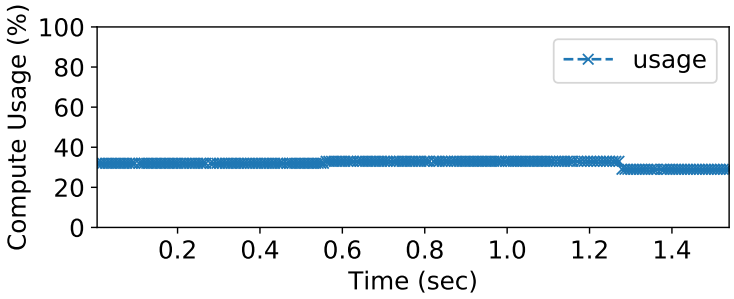


Figure 5.2: Normalized computation usage of data parallel training job using 2 V100 GPUs with gang-scheduling.

backward propagation in every model training iteration. Figure 5.2 shows that computation core is consistently underused during the whole training process due to limited on-device memory capacity.

Although fine-grained GPU sharing via multiplexing jobs on the same device can improve GPU resource utilization, it also introduce extra overheads, such as frequent context switching [41][192] and loading data from storage [42], inter-job interference [38], maintaining multiple DNN models inside the same GPU memory [40], etc. More essentially, job multiplexing schemes do not contribute to the model training process of the original job or in single job case.

In this chapter, we propose Wavelet project. Wavelet is an generic and efficient approach for achieving high resource utilization of both computation and on-device memory in order to accelerate the training process of a single job. Wavelet packs and interleaves waves of training tasks over the same set of GPUs. By interleaving peak memory usage among different

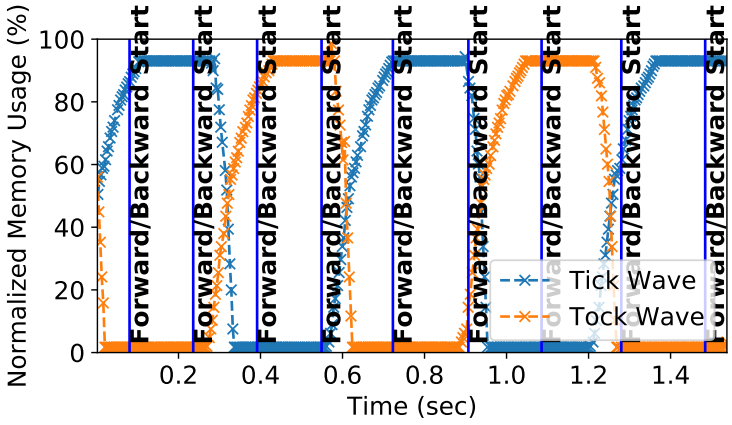


Figure 5.3: Normalized on-device memory usage of data parallel training job using 2 V100 GPUs with tick-tock scheduling.

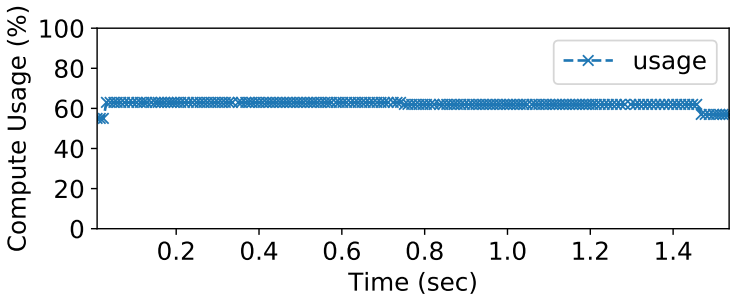


Figure 5.4: Normalized computation usage of data parallel training job using 2 V100 GPUs with tick-tock scheduling.

training waves, tasks of one wave can leverage on-device memory from tasks belonging to another wave in their memory valley periods, which increases the training throughput for a single job. As depicted in Figure 5.4 and Figure 5.3, we split data parallel training tasks into two waves: one is tick-wave and the other is tock-wave. In Figure 5.3, we intentionally delay the task launching time of tock-wave for half of the normal forward-backward training cycle. Thus, the tock-wave tasks can utilize the memory valley period of tick-wave tasks (e.g., 0.4s to 0.6s in Figure 5.3), given that the backward propagation of tick-wave tasks are computation heavy with low memory usage. Tick-wave tasks can leverage memory valley period of tock-wave tasks in the same way as we described above.

Aside from improving accelerator utilization in data parallel training, we also apply wavelet's tick-tock scheduling to model parallel training. Standard model parallel training with gang scheduling has severe resource under-utilization issue [28]. As discussed in Section 4.5 of previous chapter, the main reasons are two-fold. First, there is huge and frequent

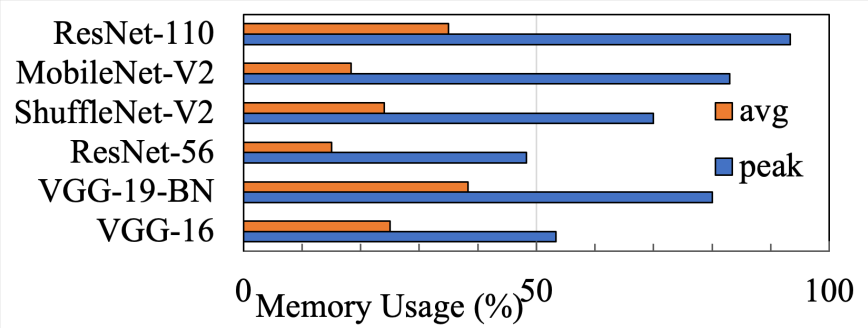


Figure 5.5: Normalized peak and average GPU memory usage during data parallel training among different CNNs.

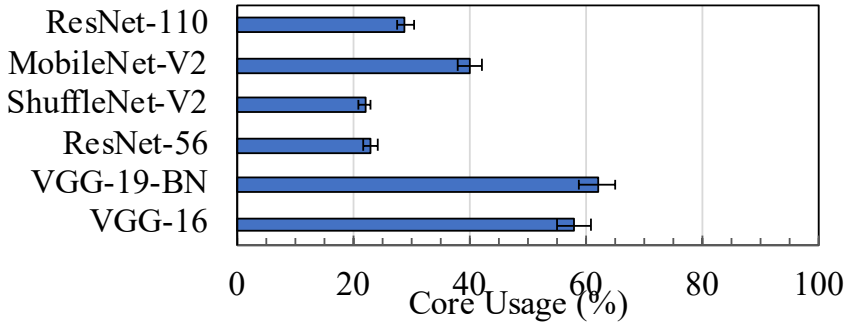


Figure 5.6: Average utilization rate of computation core during data parallel training among different CNNs.

communication overhead for transferring intermediate results among all the GPUs holding different model partitions. Second, splitting model to multiple GPUs naturally creates synchronization barriers due to DNN's sequential layer dependency [85][35]. Furthermore, even though recent literature improves vanilla model parallel training performance by incorporating pipeline parallelism [35][36], the problem is GPU under-utilization is still not fully addressed [39]. In model parallelism, given its longer memory valley periods and lower computation core utilization, we insert multiple training waves on top of the original one. As shown in Figure 5.15, based on our profiling analysis, we insert 3 additional training waves working on new batches of input data (i.e., Batch 1,2,3) on top of the original model parallel training wave (i.e., Batch 0). We arrange training tasks in round-robin fashion [193] in order to align backward propagation of same model partition across different batches of data. More details are illustrated in Section 5.3.3.

Next, we present our resource monitoring results for both data parallel and model parallel training jobs via gang-scheduling.

5.2 Motivation

In this section, we discuss the major characteristics of distributed DNN training jobs. The motivation of our work stems from the low on-device memory usage when running distributed deep learning workloads. We first present our analysis over data parallel training workload and then discuss about job attributes for model parallel training.

5.2.1 Zoom-in Analysis over Data Parallel Training

One argument we want to make is gang-scheduling, as a widely adopted design principle in distributed DNN training, may lead to system inefficiency. Here we monitor resource utilization of data parallel training jobs on several popular convolutional neural network (i.e., VGG [102], ResNet [6], MobileNet-V2 [162] and ShuffleNet-V2 [161]) using ImageNet-1K dataset [15]. We report GPU compute core and on-device memory usage for each training iteration, which includes one forward propagation followed by one backward propagation.

In Figure 5.5, it summarizes average and peak on-device memory usage in data parallel training among different CNNs. In most of the cases, the peak memory usages (i.e., blue bars in Figure 5.5) among these ConvNets training can almost reach the on-device memory capacity. However, the average memory usage among them is relatively low, which is shown as orange bars in Figure 5.5. Among all these neural nets training, the average memory usage is only around 30% of the memory capacity. Similar observation has also been found in recent literature [194][195].

In Figure 5.1, it shows a detailed spatiotemporal snapshot of GPU memory usage in the time period of 3 training iterations using ResNet-56. It indicates that the memory usage is highly predictable, and both memory valleys and peaks are well-defined in every training iteration. Figure 5.2 depicts the corresponding computation core utilization during the same period of time. As a consequence of on-device memory wall, the computation cores are consistently underutilized. The computation utilization rate is only around 30%, which leaves around 70% unused. Similar evaluation results on resource utilization are discussed in Section 5.4.1.

Figure 5.6 summarizes GPU computation utilization when training different CNNs using data parallelism. We measure GPU occupancy rate [196] as our computation utilization metric. It shows that the GPU computation cores are generally under-used, which is only around 40% on average across varied neural networks.

5.2.2 Sub-iteration Analysis on Model Parallel Training

Now we report our resource utilization results of model parallel training with gang-scheduled tasks. We collect resource utilization statistics of fine-tuning NLP model BERT [8] on SQuAD 2.0 dataset [54]. As suggested by previous literature [35][36], we split the model evenly among 4 V100 GPUs.

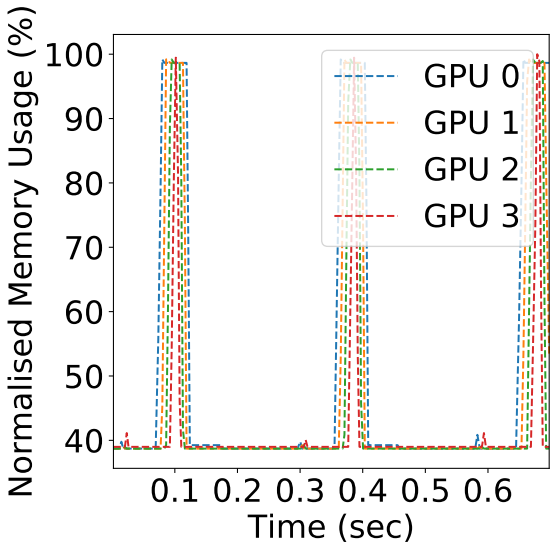


Figure 5.7: GPU Memory spatiotemporal utilization pattern of BERT model training using 4 V100 with gang-scheduled model parallelism (w/o pipeline parallelism).

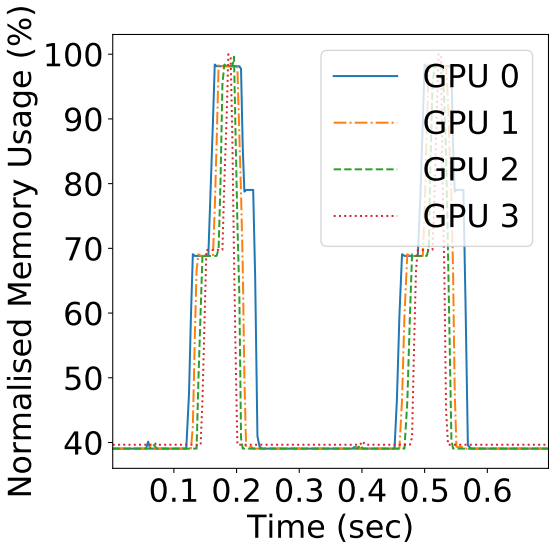


Figure 5.8: GPU Memory spatiotemporal utilization pattern of BERT model training using 4 V100 with gang-scheduled model parallelism (w/ pipeline parallelism).

As shown in Figure 5.7, Figure 5.8, Figure 5.9 and Figure 5.10, the forward propagation starts from GPU0, follows the chain of GPU0 –>GPU1 –>GPU2 –>GPU3 and ends at GPU3. The backward propagation is in the reverse direction (i.e., GPU3 –>GPU2 –>GPU1 –>GPU0). Figure 5.7 and Figure 5.8 depict the spatiotemporal memory utilization of model parallel training by incorporating pipeline parallelism (i.e., w/ pipeline parallelism) or not (i.e., w/o pipeline parallelism). For GPU computation usage, Figure 5.9 visualizes the cases without integrating pipeline parallelism and Figure 5.10 shows the cases of model parallel training with pipeline parallelism.

As shown in Figure 5.7, similar as the results we observed in data parallel training (e.g., Figure 5.1), the on-device memory usage in vanilla model parallel training also shows well-defined and clear peaks and valleys. Different from data parallel counterparts, the memory valley periods in model parallel training are much wider. This is mainly because the sequential layer dependency of DNNs generates hard synchronization barriers among GPUs holding different model partitions [85]. More precisely, the long memory valley periods are mainly due to the head-of-line blocking in the backward propagation as GPU3 –>GPU2 –>GPU1 –>GPU0. Thus, the memory valley time is much longer here than the valley period in data parallel training, which does not have data dependency during backward propagation among different GPUs. Consequently, this low on-device memory utilization leads to low computation core usage, which is shown as Figure 5.9. Results in Figure 5.8 and Figure 5.10

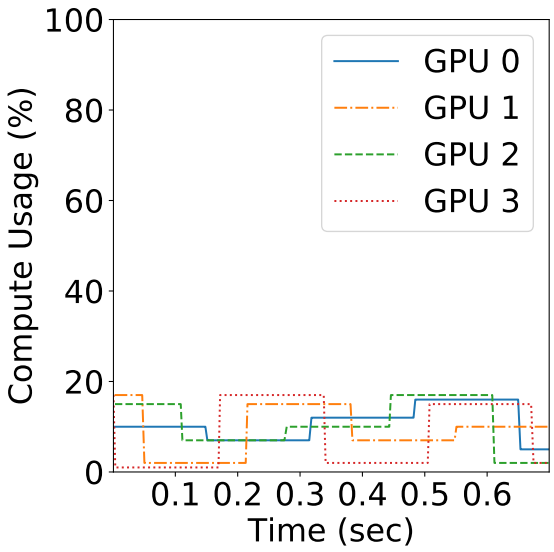


Figure 5.9: GPU computation usage of BERT model training using 4 V100 with gang-scheduled model parallelism (w/o pipeline parallelism).

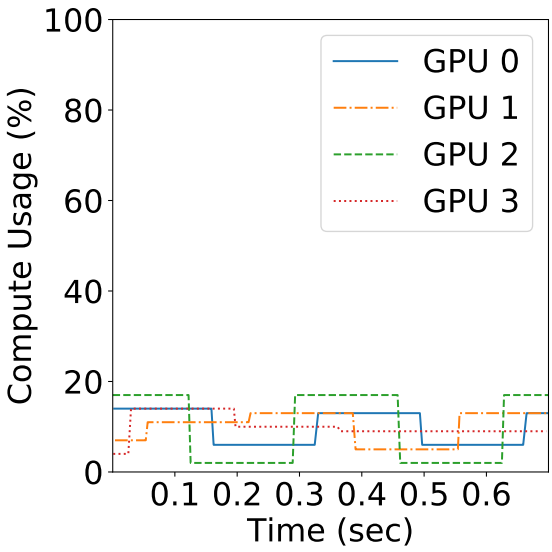


Figure 5.10: GPU computation usage of BERT model training using 4 V100 with gang-scheduled model parallelism (w/ pipeline parallelism).

indicates that, by pipelining the input data, the resource utilization is improved in model parallel training. However, this inefficiency issue is still not fully addressed. Recent literature [39] also observes similar GPU resource underutilization even by adopting optimized implementation such as Megatron-LM [28].

To sum up, this system inefficiency introduced by gang-scheduling policy is highly predictable and repeated in every training iteration for both data and model parallelism. Therefore, it leaves ample room for use to optimize and improve.

5.3 Wavelet Design

In this section, we first outline the workflow of Wavelet project. Then, we present our detailed techniques in order to address the dual challenges of achieving high utilization of both high GPU memory and computation core in data and model parallel training. We illustrate how we apply our tick-tock scheduling in data parallelism. And then we discuss how we extend it to model parallel training by injecting multiple new training waves to further improve GPU utilization and training throughput.

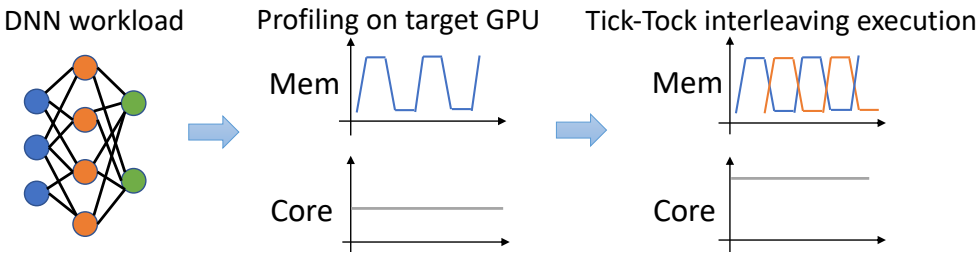


Figure 5.11: Wavelet workflow overview.

5.3.1 Overview

Given our analysis results in Section 5.2, the memory usage patterns are well-defined and repeatedly occurred across training iterations in both data parallelism and model parallelism [38][40], which provides good opportunity for further improving system efficiency.

We assume the distributed DNN training job is assigned to a group of GPUs with proper interconnects in between [93][94]. In addition, we also assume load balancing is conducted among all the accelerators, which evenly partitions the training workload among all the homogeneous GPUs. In the cases of heterogeneous environment [197][198][199][200][201], we can properly balance the workload based on each device's computation power and guarantee that they all finish each training iteration using the same amount of time. Alternatively, we can also conduct job migration to reconstruct homogeneous execution environments [38]. Since our main goal is to improve system efficiency in both GPU computation and memory usage, we directly borrow existing collective communication protocols [31][92][34] for our model synchronization and communication.

As shown in Figure 5.11, Wavelet works as follows. First, given a DNN training job, we first collects its runtime characteristics on each GPU via a short profiling over first few hundreds of mini-batch training. Given that a normal training job usually consists of millions of mini-batch training iterations [36][40], this profiling overhead is negligible.

In the profiling phase, we mainly collects three quantities for each training iteration on the target GPU. These three quantities are total runtime of single mini-batch training iteration (i.e., 1 forward + 1 backward propagation), peak and valley periods of memory usage, computation core utilization rate. With these attributes of DNN training jobs, we can determine whether it is memory-bounded computation. If so, we replace gang-scheduling with our tick-tock scheduling policy and interleave multiple training waves on each GPU. Periodically, we conduct synchronization among both intra-wave and inter-wave training tasks. By interleaving peak memory usage and packing more computation kernels into GPUs, we can achieve near optimal device utilization.

We describe our design in both data parallelism and model parallelism as following sections. We uniformly call the origial gang-scheduled training tasks as *tick-wave* tasks. And *tock-wave* tasks are the training tasks that Wavelet injects into the training pipeline via our peak memory interleaving method.

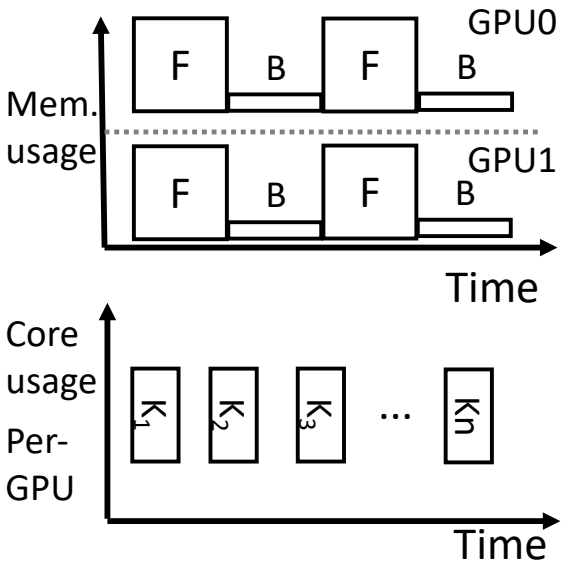


Figure 5.12: Data parallel training via gang scheduling.

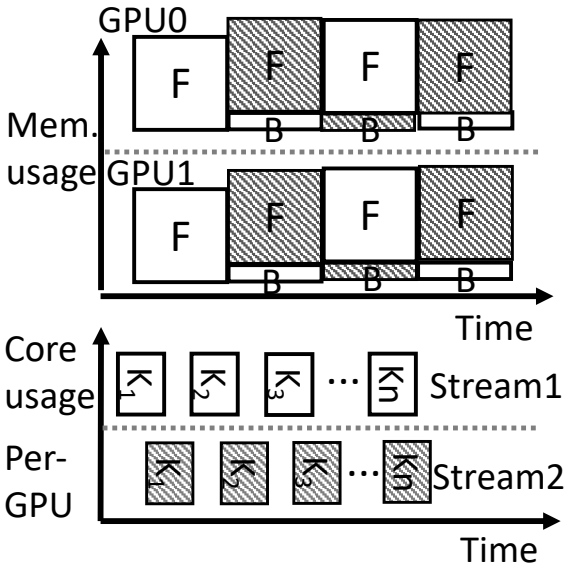


Figure 5.13: Data parallel training via tick-tock scheduling.

5.3.2 Wavelet in Data Parallelism

We now describe our Wavelet design in data parallelism. As discussed in Section 5.2.1, in data parallel training, gang-scheduling policy leaves ample room for GPU memory and computation sharing [40][202]. Figure 5.12 shows a gang-scheduled, data parallel training job with 2 GPUs, where F means forward propagation and B is backward propagation and $K_1...K_n$ are the sequence of computation kernels (i.e., K_i) launched on each GPU.

As shown in the upper of Figure 5.12, for memory usage with gang-scheduling, all the GPUs reach their memory peaks and valleys at the same time. More precisely, GPUs reach memory peaks during forward propagation and arrive at valley periods in the backward propagation. Since gang-scheduling ensures all GPUs reach their memory valley period at the same time, it is impossible for one task to leverage another task's memory valley period. Thus all the memory valley periods are wasted simultaneously during the backward propagation.

As computation utilization with gang scheduling shown in the bottom of Figure 5.12, due to limited on-device memory capacity, the tensor size for in-parallel execution inside each CUDA kernel is also limited. Therefore, the GPU computation is memory-bounded and consistently under-utilized in both forward and backward propagation.

To improve resource utilization on GPUs, a straw man approach is to interrupt and switch to another training batch during the memory valley period of tick-wave tasks (i.e., B periods in Figure 5.12). However, this preemption may block the original tick-wave training to run its own backward propagation due to the high frequency of memory peak and valley changing cycles.

In contrast to above straw man scheme, our Wavelet approach enable concurrent task launching and interleaving on the same GPU without interfering each other. We describe how we achieve this non-blocking interleaving in three aspects: memory overlapping (Section 5.3.2.1), computation overlapping (Section 5.3.2.2) and model synchronization between waves (Section 5.3.2.3).

5.3.2.1 Memory Overlapping

In contrast to gang scheduling policy, as tick-tock scheduling shown in Figure 5.13, we first allow the tick-wave tasks (i.e., blank boxes of F and B in the upper figure of Figure 5.13) to be launched as normal. Right after the tick-wave completing the forward propagation, we inject our tock-wave tasks (i.e., shadow boxes of F and B on the upper side of Figure 5.13) on to the same group of GPUs in use. This injection happens at the time when tick-wave tasks begin to free allocated memory during their backward propagation. Generally speaking, by intentionally adding task launching delay with the same duration of a normal forward propagation time, tick-tock scheduling policy can overlap tick-wave's backward propagation with tock-wave's forward propagation, and vice-versa. In addition, we do not add heavy control signals such as interrupt or preempt/resume. This simple yet decent attribute of our tick-tock scheduling guarantees we can achieve near-optimal on-device memory usage (e.g., Figure 5.3).

To simultaneously execute two training tasks (i.e., one from tick-wave, the other in tock-wave) on each GPU, we need to maintain two model replicas inside GPU memory. One replica is for maintaining model weights of tick-wave, the other replica is to hold model parameters of tock-wave. The necessity of maintaining 2 copies of model is for version control between tick and tock waves since they are training on different input data [36]. Recent literature [38][40] also reports that the popular DNN model sizes are often order-of-magnitude smaller than the generated intermediate results (e.g., activations) over input data batches.

5.3.2.2 Computation Overlapping

As mentioned in Section 5.3.2.1, we need to keep two model copies inside each GPU's memory, one for tick-wave and the other for tock-wave. Different from conventional GPU sharing among multiple jobs via time slicing [38][40], we indeed launch two different sequences of CUDA kernels concurrently on each GPU. One kernel sequence is for forward propagation of one training wave, and the other kernel sequence is for backward propagation of the other training wave. In such a way, we can improve computation core usage in both *temporal* and *spatial* dimensions.

To prevent head-of-line blocking [203] between two concurrent kernel sequences as depicted at the bottom of Figure 5.13, we launch these two sequences of kernels on different CUDA streams [105]. Thus, it guarantees the execution order of kernels within a stream, but non-blocking for CUDA kernels residing in different streams. By simultaneously launching

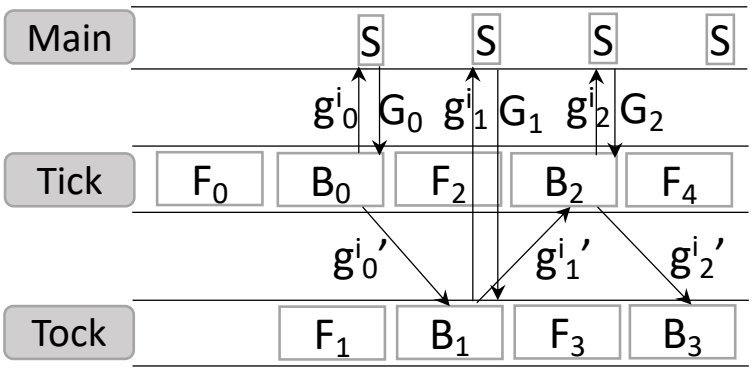


Figure 5.14: Wavelet model synchronization between tick and tock waves on GPU-i during data parallel training

multiple sequences of computation kernels on separate CUDA streams, we improve computation utilization in *spatial* dimension.

One addition benefit of simultaneously launching two streams of CUDA computation kernels is to allow kernels on different sequences to fill-in each other's execution bubble time interval. Evidence has been reported that the latency of CPU sending instructions to GPU may be amplified and create empty bubbles when launching sequences of computation kernels [71][204]. As depicted on the bottom side of Figure 5.13, the "bubble time" refers to those empty space between adjacent kernels on the same CUDA stream. Thus, by allowing kernels belonging to different CUDA streams to fill-in the "bubble time" of kernels in the same CUDA stream, Wavelet also increases the computation core usage in the *temporal* dimension.

5.3.2.3 Model Synchronization between Waves

One big component of data parallel training is model synchronization. For model synchronization among training tasks of the same wave, we can directly use existing model synchronization schemes (e.g., AllReduce) inherited from existing deep learning platforms [29][23]. Given that tasks belonging to the same wave complete each training iteration at the same time, we can directly impose model synchronization at the end of every mini-batch training iterations. The main problem here is there is no existing solution for model synchronization between our tick and tock waves on the same GPU.

Figure 5.14 shows how Wavelet enables model synchronization between tick and tock waves' training tasks. Here we have three threads (i.e., *Main*, *Tick* and *Tock*) inside the training process. In Figure 5.14, g^i_n' refers to the gradients generated on GPU i with its current local mini-batch of input. g^i_n is the averaged gradients between the $(n-1)$-th iteration and n-th iteration on GPU i, which is defined as Equation 5.1. F_n and B_n indicates the forward and backward propagation of the n-th global training iteration. S in Figure 5.14

stands for global model synchronization among all GPUs in use for the training tasks belongs to a single wave (i.e., either tick or tock wave). G_n is the global averaged gradients of the n-th iteration's model synchronization (e.g., AllReduce) among all the training tasks in the same wave.

As shown in Figure 5.14, on GPU i, After B_0 in the tick wave completes generating its local gradients as $g_0^i{\prime}$ (in the first training iteration, $g_0^i{\prime} = g_0^i$ since there is no need to average cross-wave gradients), it sends $g_0^i{\prime}$ to the subsequent tock wave task during its backward propagation (i.e., B_1). Then the $Main$ thread executes global model synchronization and get back global averaged gradient G_0 for this tick-wave training iteration. Then the task in the tick wave updates its model parameters using G_0, and begins its second training iteration as F_2. Concurrently, task B_1 on tock wave computes its local gradients as $g_1^i{\prime}$ and sends it back to tick wave's B_2 phase. Then, the tock wave B_1 averages local gradients $g_1^i{\prime}$ with previously received $g_0^i{\prime}$ from tick wave using Equation 5.1.

$$g_m^i = \frac{g_m^i{\prime} + g_{m-1}^i{\prime}}{2} \tag{5.1}$$

After gradient averaging, it uses g_1^i to do global model synchronization among all tock wave tasks from other GPUs, and get back G_1 for model updating in this tock-wave training iteration. Then, the tock wave launches its second training iteration as F_3, and so on and so forth.

To verify the model convergence, we provide following proof steps.

Lemma 1: For any globally synchronized gradients as $G_m(m > 0)$ across N GPUs, it is also globally synchronized between the overlapped one tick-wave and one tock-wave tasks (i.e., F_{m-1}, B_{m-1} and F_m, B_m in Figure 5.14).

$$G_m = \frac{\sum_{i=1}^{N}(g_{m-1}^i{\prime} + g_m^i{\prime})}{2 \times N} \tag{5.2}$$

$$\text{where } m \in \{1, 2, ...n\} \tag{5.3}$$

Proof: With Equation 5.1 for averaging local gradients between each tick-tock task pair, we directly plug-in it into Equation 5.2 and calculate the corresponding global synchronization results G_m as Equation 5.4.

$$\begin{aligned} G_m &= \frac{\sum_{i=1}^{N} g_m^i}{N} \\ &= \frac{\sum_{i=1}^{N}(g_m^i{\prime} + g_{m-1}^i{\prime})}{2 \times N} \end{aligned} \tag{5.4}$$

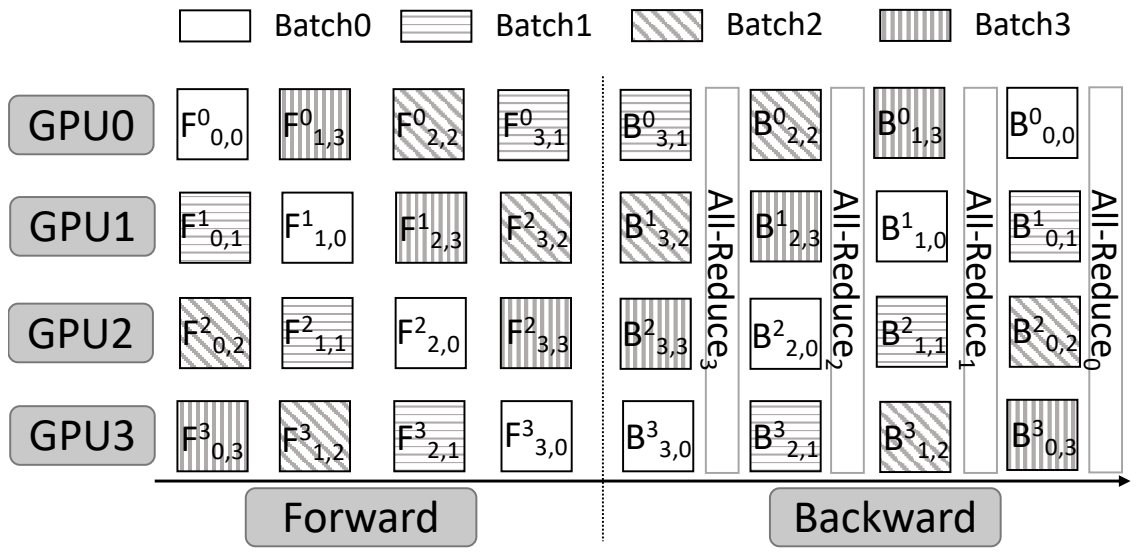


Figure 5.15: Model parallel training with Wavelet in 4-GPU setting.

Therefore, our design for model synchronization cross tick and tock waves is equivalent to model synchronization over $2 \times N$ data parallel training tasks using $2 \times N$ GPUs together, which guarantees our model convergence.

5.3.3 Wavelet in Model Parallelism

Here we discuss our model parallel design with tick-tock scheduling policy. Recently, larger models show remarkable improvement of model serving accuracy, especially in NLP domain [19][9]. State-of-the-art model serving performance for tasks like next sentence prediction and question answering is mostly achieve by transformer models, which usually contain millions or billions of model parameters [13][18].

Recent literature [35][36] leverages input batch pipelining to improving GPU resource utilization rate in model parallel training. Basically, after each GPU finishes forward computation on current micro-batch of data, rather than waiting for its turns to compute backward propagation on the same micro-batch, the GPU loads in a new micro-batch of data and conduct forward propagation. Thus by doing this data pipelining, it utilizes the idle time between 1 micro-batch's forward and backward propagation gap on each GPU.

5.3.3.1 Launching Multiple Tock-wave Tasks

As described in Section 5.2.2, the memory valley periods in model parallel training are generally longer than the data parallel counterparts. Therefore, we can leverage the wider memory valley periods on the GPUs more aggressively. Instead of injecting one tock-wave

tasks, Wavelet in model parallelism injects multiple tock-wave tasks together for training on new mini-batches of data.

Figure 5.15 shows a model parallel training job in a 4-GPU setting. Similar as our notations in Section 5.3.2, here F and B also refer to forward propagation and backward propagation. In Figure 5.15, $F^i_{m,n}$ refers to forward propagation of model partition m for data batch n on GPU-i. And $B^i_{m,n}$ follows the same naming rule. The vanilla model parallel training baseline (i.e., tick-wave) is shown as the blank boxes (e.g. $F^0_{0,0}, F^1_{1,0}, F^2_{2,0}...B^0_{0,0}$) in Figure 5.15. All the GPUs are under-utilized since at each time slot there is only one GPU working.

Different from vanilla model parallelism, we force all the GPUs to load the first model partition during the first computation cycle and execute forward propagation on different input data batches (e.g., $F^0_{0,0}, F^1_{0,1}, F^2_{0,2}, F^3_{0,3}$ in the first time slot). Similarly, in the second computation cycle, we allow all GPUs to swap to the second model partition and training on separate input data batches, and so on and so forth. Therefore, our design can be regarded as sequentially launching 3 new tock-wave tasks (i.e., tasks processing on batch 1,2,3 in Figure 5.15) on top of original tick-wave tasks (i.e., tasks working on batch 0 in Figure 5.15) in this 4-GPU case. Theoretically, we can inject $N-1$ new tock-waves with N GPUs in order to fully utilize all the GPU memory and computation resources.

5.3.3.2 Model Partition Switching

In both vanilla model parallelism and model parallelism with pipeline parallelism, each GPU always holds the same model partition during the whole training process. In contrast to both of them, Wavelet imposes each GPU to maintain different model partitions and process different input data bates in different training cycles. Therefore, frequent context switching is needed in both model partition and input data dimensions. We design our context switching scheme in a round-robin fashion [193]. As depicted in Figure 5.15, in the first forward propagation cycle, all GPUs load the first model partition and train on separate input data batches. After the first training cycle completes, we down-shift the batch-id and also up-shift the model partition number on each GPU. For instance, on GPU1, it first conduct forward propagation on batch 1 with model partition 0 in the first computation cycle (i.e., $F^1_{0,1}$). In the second compute slot, GPU 1 works on batch $(1-1)\%4 = 0$ with model partition $(0+1)\%4 = 1$, which is shown as $F^1_{1,0}$ in Figure 5.15. All the GPUs in use follows the same round-robin swapping on both data and model partitions. And backward propagation is just symmetric too the forward propagation but in the inverse order.

We note that frequent context switching of both input data and model partitions would definitely introduce overheads. We empirically evaluate our context switching overheads via end-to-end DNN training experiments in Section 5.4.2. To further reduce our on-device memory overheads for holding intermediate results, we can leverage the techniques such as tensor re-materialization [205][206][207] to recompute the intermediate results on-demand. And we leave it as one of our future research directions.

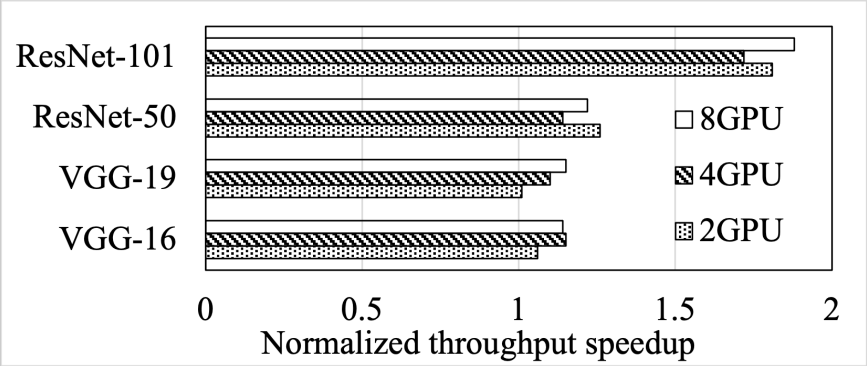


Figure 5.16: Wavelet's throughput speedup over data parallel training baseline (single-machine multi-GPU).

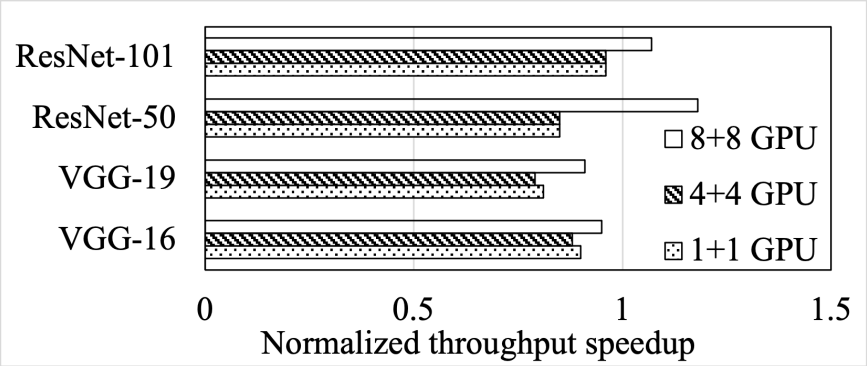


Figure 5.17: Wavelet's throughput speedup over data parallel training baseline (multi-machine multi-GPU).

5.3.3.3 Inter-batch Synchronization

In Figure 5.15, with our round-robin scheme, we inject $(N-1)$ tock waves on top of the original tick-wave training in N GPUs setting. With our round-robin assignments, for each single model partition, we can generate gradients from all the batches at the same time. For example, in Figure 5.15, the backward propagation for model partition 3 across batches of 0,1,2,3 (i.e., $B^x_{3,y}$, $x, y \in \{0, 1, 2, 3\}$) are finished at the same time. This inherit alignment of backward propagation across batches on the same model partition simplifies our model synchronization design in model parallel training. Here we directly impose AllReduce right after the backward propagation of same model partition across separate batches, and update the model parameters for each model partition accordingly (e.g., All-Reduce$_3$ for $B^x_{3,y}$).

5.4 Evaluation

We evaluate Wavelet performance using five different DNNs on two kinds of datasets. For data parallel training, we test Wavelet on several popular CNNs. The ones we use are VGG-16 [102], VGG-19 with batch normalization [58], ResNet-50 and ResNet-101 [6]. We train these models on ImageNet-1K dataset [7]. For model parallel training, we first fine-tune BERT model [8] with SQuAD2.0 dataset [54]. We also test model parallel training with VGG-19 on ImageNet-1K dataset. All the experiments are executed on DGX-1 [93] machines, which contains V100 GPUs and point-to-point NVLink connection among the GPUs within a machine. For cross-machine communication, the link in between is 25Gb/s Ethernet. We test both single-machine multi-GPU cases and multi-machine multi-GPU cases for both data parallel and model parallel training experiments.

We report a number of important findings from our evaluation. First, Wavelet can achieve up to 1.88x throughput speedup over data parallel training baseline. Second, Wavelet outperforms over hybrid of model and pipeline parallel training by up to 4.15x speedup. We achieve up to 6.7x faster in training when compared with vanilla model parallel training baseline.

5.4.1 Data Parallelism

In our data parallel training setting, we use the biggest mini-batch size that can fit into the GPU memory. This guarantees the highest throughput can be reached in our baseline. It also ensures the best GPU memory utilization that Wavelet can achieve via our tick-tock interleaving. Here we choose 64 as the per-GPU mini-batch size for data parallel training on both VGGNets and ResNets.

5.4.1.1 Single-machine Multi-GPU

In single-machine multi-GPU case, we test follow three different settings: data parallel training with 2*V100 GPUs, 4*V100 GPUs and 8*V100 GPUs. All these GPUs are connected with NVLink, which provides around 25GB/s per link.

We normalize the throughput of data parallel training baseline as 1, and report our speed-up numbers over the normalized baseline. As depicted in Figure 5.16, Wavelet achieves up to 1.88x speed up over the data parallel baseline. The average training throughput gain is 1.4x across different CNNs and different number of GPUs in use. Note that we can only achieve <1.2x speedup for some VGG training, this is mainly because each training iteration time of VGGs are short. Since our injection of tock-wave tasks lengthen the duration of each training iteration, the speedup number is inversely proportional to original training iteration time. Thus, if the original training iteration time is short, our performance gain is less pronounce, and vice versa.

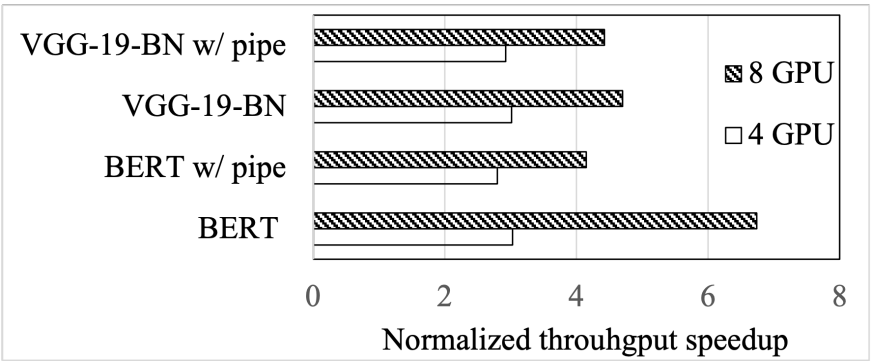


Figure 5.18: Wavelet's throughput speedup over model parallel training baseline (single-machine multi-GPU).

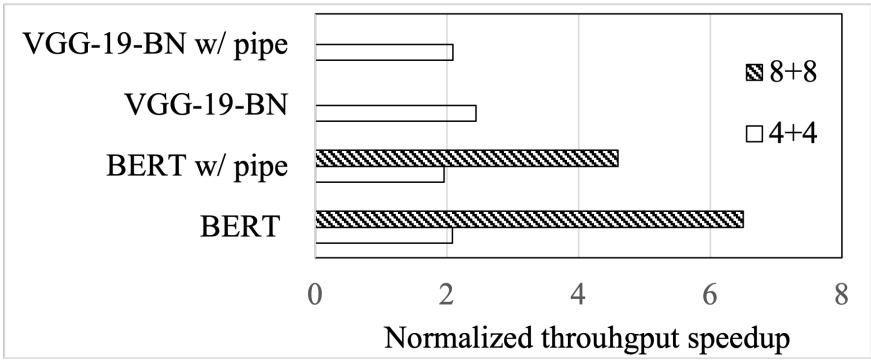


Figure 5.19: Wavelet's throughput speedup over model parallel training baseline (multi-machine multi-GPU).

5.4.1.2 Multi-machine Multi-GPU

In multi-machine multi-GPU settings, we use two DGX-1 machine together. We test the following three different situations: $8+8$, $4+4$ and $1+1$. In Figure 5.17, $1+1$ means we use 1 GPU on one machine, and another GPU on the other machine. $8+8$ and $4+4$ follow the same naming rule.

In contrast to single-machine performance, here we can only achieve up to 1.18x speedup over data parallel training baseline in these multi-machine cases. Additionally, our Wavelet performs even worse than the baseline in many cases in Figure 5.17. This is mainly due to the limited cross-machine network bandwidth. Since our inserted tock-wave needs additional cross-machine model synchronization (e.g., AllReduce), it adds extra latency in each training iteration. Although we also interleave AllReduce operations between tick-wave and tock-wave tasks (e.g., G_0, G_1 in Fig. 5.14) to mitigate the burst of communication, the limited inter-machine bandwidth is still the major throughput bottleneck.

5.4.2 Model Parallelism

In our model parallel training setting, as suggested by previous literature [35][36], we evenly split both VGG-19 and BERT models among all the GPUs in use for better load balancing. Similar as our data-parallel setups, we also choose the largest batch size per-GPU without introducing out-of-memory exceptions.

5.4.2.1 Single-machine Multi-GPU

In single machine case, we test 4-GPU and 8-GPU cases of model parallel training on VGG-19 and BERT. For each model we test two different settings: one is vanilla model parallelism and the other is model parallelism with pipeline parallelism (i.e., w/ pipe in Figure 5.18). In Figure 5.18, Wavelet outperforms baseline with higher throughput speedup number in 8-GPU cases than 4-GPU cases. More precisely, compared with vanilla model parallel baseline, we achieve 4.7x speedup on VGG-19 and 6.7x on BERT in 8-GPU settings. Comparing to model parallel training with Pipeline parallelism, Wavelet achieves 4.4x speedup on VGG-19 and 4.15x speed up on BERT in 8-GPU cases.

Even by incorporating pipeline parallelism such as Gpipe [35], the reported speedups over vanilla model parallel baseline is normally <2x with 4-GPU and around 3x in 8-GPU settings [35]. This is mainly due to the high frequency in both CUDA kernel launching and intermediate results communication. More precisely, regarding each mini-batch (e.g., of size M) of input data, Gpipe or PipeDream [36] further breaks it into multiple small micro-batches (e.g., of size N) and pipelines training over micro-batches. Therefore, for training 1 single mini-batch data, the frequency of communication increases from 2 (i.e., for each GPU, 1 for forward propagation, 1 for backward propagation) to $2 \times \frac{M}{N}$ (i.e., first $\frac{M}{N}$ for forward propagation, and second $\frac{M}{N}$ for backward propagation). More crucially, this high frequent but small data chunks may not fully saturate the link bandwidth such as NVLink [88] and NVSwitch [90], which leads to longer communication latency. Similarly, the number of CUDA kernel calls also increase by $\frac{M}{N}$ times for processing 1 mini-batch input, which also introduces higher control overheads. In contrast, for 1 mini-batch training, Wavelet still maintains the communication frequency of 2 with same number of CUDA kernels as vanilla model parallel training. Therefore, we can achieve higher system throughput compared with pipeline parallelism.

5.4.2.2 Multi-machine Multi-GPU

As shown in Figure 5.19, due to low bandwidth cross-machine interconnects, we achieve only moderate (i.e. 2x) speedup over baseline with or without incorporating pipeline parallelism in our 4+4 cross-machine cases. For 8+8 settings, we only test with BERT model, since it is unreasonable to split a ConvNet over 16 GPUs. In 8+8 setting, we achieve 6.5x speedup over vanilla model parallel training, and 4.5x faster over BERT training with pipeline parallelism.

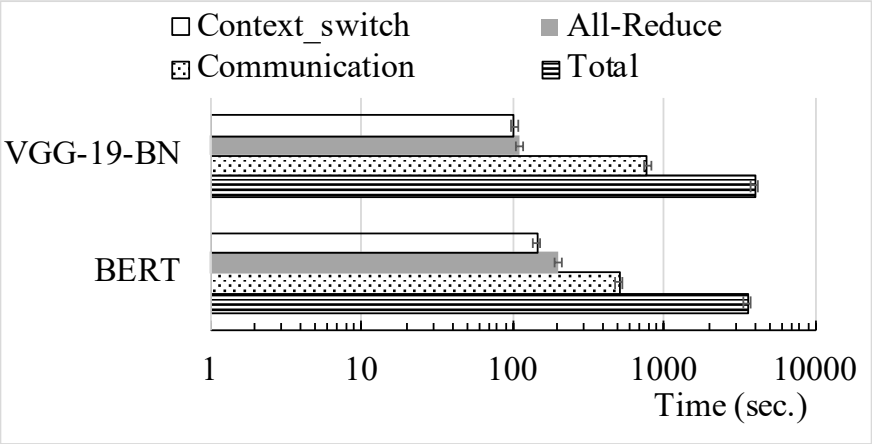


Figure 5.20: Wavelet overhead breakdown in the 4+4 cross-machine case

Given that our task pipelining only happens on each single GPU, we believe increasing the number of GPUs to very large systems will not cause any significant performance degradation.

5.4.2.3 Overhead Analysis

Now we analyze the overheads introduced by Wavelet's tock-wave injections in model parallel training. Here we only test one hardware setting which is $4+4$, which means we use 4 GPUs on one machine, and another 4 GPUs on the other machine.

In Figure 5.20, we record the total time for training one epoch on both VGG-19 and BERT models. We also collect the time breakdowns in three major overheads as *context_switch*, *communication* and *All-Reduce*. Here *context_switch* denotes the time spending on switching model partitions in different model training cycles. The *communication* is the time for transmitting intermediate results (e.g., gradients) among all the GPUs involved in the same job. The *All-Reduce* overhead refers to model synchronization time for each model partition across different data batches in the backward propagation.

As shown in Figure 5.20, the cross-machine *communication* takes longest time duration among all these main system overheads. *Communication* takes 19.5% of total iteration training time in VGG-19, and 14% of iteration time in BERT training. The sum of *All-Reduce* and *context_switch* only occupies 5% iteration time on BERT and 2% on VGG-19, separately. Therefore, the major system overhead is caused by low bandwidth cross-machine networks, not *context_switch* or model synchronization (i.e. *All-Reduce*) time introduced by Wavelet.

5.5 Related Work

Previous literature mainly falls into the two categories below.

Resource allocation in distributed DNN training: A number of resource allocation and scheduling policies are tailor-made for distributed DNN training jobs [51]. They may have varied focus such as job fairness [49], better overlapping between computation and communication [208], better job locality [38], improving system goodput [209], minimizing job completion time [50], transparent auto-scaling [191] [194], etc. However, at the in-parallel task launching stage, gang-scheduling is used by-default and could be the *only* option for task launching of distributed DNN training job. In contrast, Wavelet proposes tick-tock policy as another option for in-parallel task launching of distributed ML workloads. Comparing with gang-scheduling, our tick-tock scheduling achieves higher GPU computation and memory usage. Consequently, we boost up the training speed in single job case.

GPU sharing: Several recent work has been focusing on fine-grained GPU sharing, such as Nvidia's MPS [204], Salus [40], Gandiva [38], etc. CUDA Multi-Process Service (MPS) is a runtime architecture which allows multiple CUDA processes to share the same GPU memory with static partitioning. When multiplexing different jobs on to the same GPU, static memory partitioning introduces frequent inter-job interference, which may leads to job failure and decreases overall performance. Gandiva co-locates deep learning jobs first and conducts job migration to run each job within minimum number of machines for better data locality, and finally reach no-sharing mode. Salus achieves fine-grained GPU sharing among multiple deep learning jobs by sharing computation resource in spatiotemporal way. Nvidia's TensorRT [71] supports concurrent deep learning model inference on the same GPU, but lack of training supports. Different from all the works listed above targeting on multiplexing multiple jobs on same device, Wavelet allows GPU sharing among different training waves of a single job, thus improves the training speed in single job case.

Earlier literature on GPU sharing [210][211][212] focus on workloads with just a few CUDA kernel functions. Thus, it could not be directly applicable to deep learning applications with hundreds of unique kernel executions.

5.6 Summary

In this chapter, we present a novel way to improve training throughput of single job via our tick-tock scheduling policy. By interleaving peak memory usage among different training waves of a single job, we improve GPU utilization rate in both computation and on-device memory aspects. By replacing gang-scheduling with our tick-tock scheduling policy, Wavelet achieves up to 6.7x time reduction of each training iteration in single job case. Wavelet is generic and can be applied to data parallel, model parallel and hybrid DNN training paradigms.

Chapter 6

Future Work and Conclusion

6.1 Future Directions

Go distributed or go centralized, that is the question.

Clearly, given giant deep learning models and massive data, practitioners naturally go distributed with more accelerators. Countless research papers have been published to improve system efficiency under various distributed settings. However, low cross-machine bandwidth and limited on-device memory already set the upper bound on system throughput.

Recently, we see the trend that hardware manufacturers are trying to pack more and more accelerators into a single box. For example, Nvidia groups 16 GPUs inside a single server with fast interconnects in between [94]. From Apple's original M1 chip, to M1 Pro/Max and now M1 Ultra, they build larger System-on-a-chip (SoC) by packing more and more GPU cores on a single chip [213]. Maybe it is time to design smart scheduling algorithms in these centralized settings.

6.2 Concluding Remarks

This dissertation proposes novel designs to efficiently distribute machine learning workloads.

We first illustrate why distributed model training and serving are needed as our thesis motivation. Following that we highlight two major issues when scaling out: high communication overheads and limited on-device memory. Before diving into our solutions, we provide background knowledge on deep learning models and popular parallel paradigms.

In Blink, we argue that packing spanning trees rather than forming rings achieves better link utilization given arbitrary network environments. In sensAI, by decoupling a base model into disconnected subnets via our class parallelism, we nearly eliminate the communication in model parallelism. In Wavelet, by intentionally adding task launching latency, we increase both GPU utilization and training throughput in single job case.

Bibliography

[1] Andrew Ng, *Coursera: Machine learning course at stanford*, https://www.coursera.org/learn/machine-learning, 2022.

[2] Ethem Alpaydin, *Machine learning*. MIT Press, 2016.

[3] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.

[4] MIT Technology Review, *The fourth industrial revolution has begun: Now's the time to join*, https://www.technologyreview.com/2020/10/15/1010365/the-fourth-industrial-revolution-has-begun-nows-the-time-to-join/, 2020.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, 2012.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, "Language models are unsupervised multitask learners," 2019.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, 2017.

[12] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollar, "Microsoft coco: Common objects in context," *arXiv preprint arXiv:1405.0312*, 2015.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei, "Language models are few-shot learners," in *NeurIPS*, 2020.

[14] Alex Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, 2015.

[16] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, "Going deeper with convolutions," *arXiv preprint arXiv:1409.4842*, 2014.

[17] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le, "Regularized evolution for image classifier architecture search," in *AAAI*, 2019.

[18] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[19] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever, "Zero-shot text-to-image generation," *arXiv preprint arXiv:2102.12092*, 2021.

[20] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng, "Large scale distributed deep networks," in *NeurIPS*, 2012.

[21] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[22] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," in *VLDB*, 2020.

[23] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "Tensorflow: A system for large-scale machine learning," in *USENIX OSDI*, 2016.

[24] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su, "Scaling distributed machine learning with the parameter server," in *USENIX OSDI 2014*, 2014.

[25] Guanhua Wang, *Distributed Machine Learning with Python: Accelerating model training and serving with distributed systems*. Packt, 2022.

[26] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *NeurIPS*, 2014.

[27] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing, "Strads: A distributed framework for scheduled model parallel machine learning," in *EuroSys*, 2016.

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

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "PyTorch: an imperative style, high-performance deep learning library," in *NeurIPS*, 2019.

[30] Frank Seide and Amit Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD'16, 2016.

[31] Sylvain Jeaugey, *Optimized inter-GPU collective operations with NCCL 2*, https://developer.nvidia.com/nccl, 2017.

[32] Alex Sergeev and Mike Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[33] Pieter Noordhuis, *Accelerating machine learning for computer vision*, https://github.com/facebookincubator/gloo, 2017.

[34] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica, "Blink: Fast and Generic Collectives for Distributed ML," in *Third Conference on Machine Learning and Systems (MLSys)*, 2020.

[35] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *NeurIPS*, 2019.

[36] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[37] Guanhua Wang, Zhuang Liu, Brandon Hsieh, Siyuan Zhuang, Joseph Gonzalez, Trevor Darrell, and Ion Stoica, "sensAI: ConvNets Decomposition via Class Parallelism for Fast Inference on Live Data," in *Fourth Conference on Machine Learning and Systems (MLSys)*, 2021.

[38] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[39] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler, "Data Movement Is All You Need: A Case Study on Optimizing Transformers," in *Fourth Conference on Machine Learning and Systems (MLSys 2021)*, 2021.

[40] Peifeng Yu and Mosharaf Chowdhury, "Salus: Fine-grained gpu sharing primitives for deep learning applications," in *Third Conference on Machine Learning and Systems (MLSys)*, 2020.

[41] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[42] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram, "Analyzing and mitigating data stalls in dnn training," in *VLDB*, 2021.

[43] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica, "Wavelet: Efficient DNN Training with Tick-Tock Scheduling," in *Fourth Conference on Machine Learning and Systems (MLSys)*, 2021.

[44] Nvidia, *Graphics processing unit*, https://www.nvidia.com/en-us/geforce/graphics-cards/, 2022.

[45] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon,

James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[46] Wikipedia-fpga, *Field-programmable gate array*, https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[47] Wikipedia-asic, *Application-specific integrated circuit*, https://en.wikipedia.org/wiki/Application-specific_integrated_circuit.

[48] Beidi Chen, Tharun Medini, Sameh Gobriel James Farwell, Charlie Tai, and Anshumali Shrivastava, "Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," in *MLSys*, 2020.

[49] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla, "Themis: Fair and efficient gpu cluster scheduling," in *USENIX NSDI*, 2020.

[50] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[51] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *USENIX ATC*, 2019.

[52] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NeurIPS*, 2015.

[53] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014.

[54] Pranav Rajpurkar, Robin Jia, and Percy Liang, "Know what you don't know: Unanswerable questions for squad," in *Annual Meeting of the Association for Computational Linguistics(ACL)*, 2018.

[55] Jens Kober, J. Andrew Bagnell, and Jan Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, 2013.

[56] Yann LeCun, Koray Kavukcuoglu, and Clement Farabet, "Convolutional networks and applications in vision," in *ISCAS*, 2010.

[57] Fei-fei Li, *Deep learning for computer vision*, http://cs231n.stanford.edu/, 2022.

[58] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of Machine Learning Research*, 2015.

[59] Zhongxia Yan, Jingguo Ge, Yulei Wu, Liangxiong Li, and Tong Li, "Automatic virtual network embedding: A deep reinforcement learning approach with graph convolutional networks," *IEEE Journal on Selected Areas in Communications*, 2020.

[60] Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He, "Recurrent reinforcement learning: A hybrid approach," *arXiv:1509.03044*, 2015.

[61] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," in *NeurIPS*, 2021.

[62] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, Matthew Botvinick, Nicolas Heess, and Raia Hadsell, "Stabilizing transformers for reinforcement learning," in *Proceedings of the 37th International Conference on Machine Learning (PMLR)*, 2020.

[63] Hugging Face, *Tokenizer*, https://huggingface.co/docs/transformers/main_classes/tokenizer, 2022.

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

[65] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, G. Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou, "Mlperf inference benchmark," in *ISCA*, 2020.

[66] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica, "Clipper: A low-latency online prediction serving system," in *USENIX NSDI*, 2017.

[67] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, Mostafa El-Khamy, Jungwook Hong, Tom St John, Cindy Trinh, Michael Buch, Mark Mazumder, Relja Markovic, Thomas Atta, Fatih Cakir, Masoud Charkhabi, Xiaodong Chen,

Cheng-Ming Chiang, Dave Dexter, Terry Heo, Guenther Schmuelling, Maryam Shabani, and Dylan Zika, "Mlperf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device ai," in *MLSys*, 2022.

[68] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov, "Inferline: Ml inference pipeline composition framework," in *ACM SoCC*, 2020.

[69] Google, *Tfrt: A new tensorflow runtime*, https://blog.tensorflow.org/2020/04/tfrt-new-tensorflow-runtime.html, 2020.

[70] onnxruntime, https://onnxruntime.ai/pytorch, 2022.

[71] Nvidia, *Tensorrt*, https://developer.nvidia.com/tensorrt, 2022.

[72] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora, "Communication-efficient federated learning with sketching," in *ICML*, 2020.

[73] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo, "A generic communication scheduler for distributed dnn training acceleration," in *ACM SOSP*, 2019.

[74] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[75] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *MLSys*, 2019.

[76] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *EuroSys*, 2016.

[77] Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Olli Saarikivi, Emad Barsoum, Jaliya Ekanayake, Vadim Eksarevskiy, and Tianju Xu, "Scaling distributed training with adaptive summation," in *MLSys*, 2021.

[78] Nadeen Gebara, Paolo Costa, and Manya Ghobadi, "In-network aggregation for shared machine learning clusters," in *MLSys*, 2021.

[79] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," in *MLSys*, 2019.

[80] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar, "PowerAI DDL," *arXiv preprint arXiv:1708.02188*, 2017.

[81] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia, "Mlperf training benchmark," in *MLSys*, 2020.

[82] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[83] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[84] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *USENIX OSDI*, 2022.

[85] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," in *ICLR*, 2021.

[86] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, Santa Clara, CA: USENIX Association, 2017, pp. 181–193.

[87] *PCI Express: An Overview of the PCI Express Standard*, http://www.ni.com/white-paper/3767/en/, 2014.

[88] *NVIDIA NVLINK*, http://www.nvidia.com/object/nvlink.html, 2017.

[89] *Introduction to InfiniBand*, https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf, 2007.

[90] *NVIDIA NVSWITCH*, http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018.

[91] "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.

[92] Blaise Barney, *Message Passing Interface*, https://computing.llnl.gov/tutorials/mpi/, 2018.

[93] *NVIDIA DGX-1*, https://www.nvidia.com/en-us/data-center/dgx-1/, 2020.

[94] *NVIDIA DGX-2*, https://www.nvidia.com/en-us/data-center/dgx-2/, 2021.

[95] *NVIDIA Tesla P100 GPU*, https://www.nvidia.com/en-us/data-center/tesla-p100/, 2018.

[96] *NVIDIA V100 GPU*, https://www.nvidia.com/en-us/data-center/v100/, 2019.

[97] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang, "Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications," *Microsoft Research Technical Report (MSR-TR-2018-13)*, 2018.

[98] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu, "Allox: Compute allocation in hybrid clusters," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[99] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.

[100] *NVIDIA A100 GPU*, https://www.nvidia.com/en-us/data-center/a100/, 2021.

[101] *NVIDIA H100 GPU*, https://www.nvidia.com/en-us/data-center/h100/, 2022.

[102] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[103] Laszlo Lovasz, "On two minimax theorems in graph," *Journal of Combinatorial Theory, Series B*, vol. 21, no. 2, pp. 96–103, 1976.

[104] Jack Edmonds, "Edge-disjoint branchings," *Combinatorial algorithms*, 1973.

[105] *NVIDIA CUDA toolkit*, https://developer.nvidia.com/cuda-toolkit, 2022.

[106] Harold N Gabow and KS Manu, "Packing algorithms for arborescences (and spanning trees) in capacitated graphs," *Mathematical Programming*, vol. 82, no. 1-2, pp. 83–109, 1998.

[107] Chandra Chekuri and Kent Quanrud, "Near-linear time approximation schemes for some implicit fractional packing problems," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2017, pp. 801–820.

[108] Pitch Patarasuk and Xin Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, pp. 117–124, 2009.

[109] *Massively Scale Your Deep Learning Training with NCCL 2.4*, https://bit.ly/2lFwFQ4, 2019.

[110] *Wiki of TCP/IP Internet protocol suite*, https://en.wikipedia.org/wiki/Internet_protocol_suite, 2022.

[111] Shelby Thomas, Geoffrey M. Voelker, and George Porter, "Cachecloud: Towards speed-of-light datacenter communication," in *USENIX hotcloud 2018*, 2018.

[112] *Removing roadblocks on the path to 400G and beyond*, https://bit.ly/2k4PXh9, 2018.

[113] *Verizon marks milestone with successful 400G technology trial*, https://bit.ly/2lKgAs7, 2018.

[114] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn, "Global combine on mesh architectures with wormhole routing," in *Proceedings of the 7th International Parallel Processing Symposium*, 1993.

[115] S. Bokhari and H. Berryman, "Complete exchange on a circuit switched mesh," in *Proceedings of the Scalable High Performance Computing Conference*, 1992.

[116] D. Scott, "Efficient all-to-all communication patterns in hypercube and mesh topologies," in *Proceedings of the 6th Distributed Memory Computing Conference*, 1991.

[117] Laxmi N. Bhuyan and Dharma P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, 1984.

[118] S.L. Johnsson and C.-T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *IEEE Transactions on Computers*, 1989.

[119] Rajeev Thakur, Rolf Rabenseifner, and William Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, 2005.

[120] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra, "Automatically tuned collective communications," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00, 2000.

[121] Huasha Zhao and John Canny, "Butterfly mixing: Accelerating incremental-update algorithms on clusters," in *Proceedings of the 2013 SIAM International Conference on Data Mining*, 2013.

[122] Rolf Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*, 2004.

[123] Robert van de Geijn, "On global combine operations," in *Journal of Parallel and Distributed Computing*, 1994.

[124] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan, "Bandwidth efficient all-to-all broadcast on switched clusters," *International Journal of Parallel Programming*, 2008.

[125] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium*, ser. IEEE IPDPS'00, 2000.

[126] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. ACM PPoPP'99, 1999.

[127] Yifan Gong, Bingsheng He, and Jianlong Zhong, "Network performance aware mpi collective communication operations in the cloud," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

[128] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe, "Machine-aware atomic broadcast trees for multicoress," in *USENIX OSDI*, 2016.

[129] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[130] Zhihao Jia, Matei Zaharia, and Alex Aiken, "Beyond data and model parallelism for deep neural networks," in *MLSys*, 2019.

[131] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *arXiv preprint arXiv:1512.01274*, 2015.

[132] Brian Paden, Michal Cap, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[133] Claudine Badue, Ranik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius Brito Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago Paixao, Filipe Mutz, Lucas Veronese, Thiago Oliveira-Santos, and Alberto Ferreira De Souza, "Self-driving cars: A survey," in *arXiv:1901.04407*, 2019.

[134] Daniel G. Jennings, *Goldman Sachs Gambles Big in AI*, https://bit.ly/3jNLkCj, 2018.

[135] Jeremy Horwitz, *Goldman: AI tools have potential in finance beyond smart stock trading*, https://bit.ly/3ckkX3A, 2020.

[136] Katia Porzecanski, *JPMorgan Commits Hedge Fund to AI in Technology Arms Race*, https://bloom.bg/2P1JUpF, 2019.

[137] Fuxun Yu, Zhuwei Qin, and Xiang Chen, "Distilling Critical Paths in Convolutional Neural Networks," in *NeurIPS CDNNRIA workshop*, 2018.

[138] Bolei Zhou, Yiyou Sun, David Bau, and Antonio Torralba, "Revisiting the importance of individual units in cnns via ablation," *arXiv preprint arXiv:1806.02891*, 2018.

[139] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *NeurIPS*, 2014.

[140] Jimmy Ba and Rich Caruana, "Do deep nets really need to be deep?" In *NeurIPS*, 2014.

[141] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang, "Learning efficient convolutional networks through network slimming," in *ICCV*, 2017.

[142] Song Han, Jeff Pool, John Tran, and William Dally, "Learning both weights and connections for efficient neural network," in *NeurIPS*, 2015.

[143] Song Han, Huizi Mao, and William J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[144] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf, "Pruning filters for efficient convnets," in *ICLR*, 2017.

[145] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell, "Rethinking the value of network pruning," in *ICLR*, 2019.

[146] Arnav Chavan, Zhiqiang Shen, Zhuang Liu, Zechun Liu, Kwang-Ting Cheng, and Eric Xing, "Vision transformer slimming: Multi-dimension searching in continuous optimization space," in *CVPR*, 2022.

[147] Mikel Galara, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera, "An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes," *Pattern Recognition*, vol. 44, pp. 1761–1776, 2011.

[148] Alina Beygelzimer, Hal Daume III, John Langford, and Paul Mineiro, "Learning reductions that really work," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 136–147, 2016.

[149] Ryan Rifkin and Aldebaro Klautau, "In defense of one-vs-all classification," *Journal of Machine Learning Research*, vol. 5, pp. 101–141, 2004.

[150] Alina Beygelzimer, John Langford, and Bianca Zadrozny, "Weighted one-against-all," in *AAAI*, 2005.

[151] Thomas G. Dietterich and Ghulum Bakiri, "Solving multiclass learning problems via error-correcting output codes," *Journal of Artificial Intelligence Research*, vol. 2, pp. 263–286, 1995.

[152] Eun Bae Kong and Thomas G. Dietterich, "Error-correcting output coding corrects bias and variance," in *ICML*, 1995.

[153] Huiqun Deng, George Stathopoulos, and Ching Y. Suen, "Applying error-correcting output coding to enhance convolutional neural network for target detection and pattern recognition," in *International Conference on Pattern Recognition*, 2010.

[154] Rangachari Anand, Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka, "Efficient classification for multiclass problems using modular neural networks," *IEEE Transactions on Neural Networks*, vol. 6, no. 1, pp. 117–124, 1995.

[155] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang, "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures," *arXiv preprint arXiv:1607.03250*, 2017.

[156] Laurens van der Maaten and Geoffrey Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, 2008.

[157] George C. Linderman and Stefan Steinerberge, "Clustering with t-sne, provably," *SIAM Journal on Mathematics of Data Science*, vol. 1, 2019.

[158] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, University of California Press, 1967, pp. 281–297.

[159] Mikko I. Malinen and Pasi Franti, "Balanced k-means for clustering," in *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, 2014.

[160] Josh Levy-Kramer and Matt Klaber, *k-means-constrained 0.4.3*, https://pypi.org/project/k-means-constrained/, 2020.

[161] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *CVPR*, 2018.

[162] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[163] Nvidia, *Data sheet: Tesla m60*, https://images.nvidia.com/content/tesla/pdf/188417-Tesla-M60-DS-A4-fnl-Web.pdf, 2016.

[164] Yuxin Wu and Kaiming He, "Group normalization," in *ECCV*, 2018.

[165] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie, "A convnet for the 2020s," in *CVPR*, 2022.

[166] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari, "Accelerating 3d deep learning with pytorch3d," *arXiv preprint arXiv:2007.08501*, 2020.

[167] Jianbo Ye, Xin Lu, Zhe Lin, and James Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers," in *ICLR*, 2018.

[168] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz, "Pruning convolutional neural networks for resource efficient inference," in *ICLR*, 2017.

[169] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *IJCAI*, 2018.

[170] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *CVPR*, 2019.

[171] Michael Zhu and Suyog Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[172] Yi Wei, Xinyu Pan, Hongwei Qin, Wanli Ouyang, and Junjie Yan, "Quantization mimic: Towards very tiny cnn for object detection," in *ECCV*, 2018.

[173] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[174] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng, "Quantized convolutional neural networks for mobile devices," in *CVPR*, 2016.

[175] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of The 33rd International Conference on Machine Learning (PMLR)*, 2016.

[176] Ron Banner, Yury Nahshan, and Daniel Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," in *NeurIPS*, 2019.

[177] Wei Yang, *pytorch-classification on CIFAR10/100 and ImageNet*, https://github.com/bearpaw/pytorch-classification, 2017.

[178] D J. Baylis, *Error Correcting Codes: A Mathematical Introduction*. Chapman and Hall/CRC, 1997.

[179] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman, "Parity models: Erasure-coded resilience for prediction serving systems," in *SOSP*, 2019.

[180] Wikipedia, *Cyclic code*, https://en.wikipedia.org/wiki/Cyclic_code.

[181] Yaoqing Yang, Jichan Chung, Guanhua Wang, Vipul Gupta, Adarsh Karnati, Kenan Jiang, Ion Stoica, Joseph Gonzalez, and Kannan Ramchandran, "Robust Class Parallelism - Error Resilient Parallel Inference with Low Communication Cost," in *the 54th Asilomar Conference on Signals, Systems, and Computers (Asilomar 2020)*, 2020.

[182] Fangyu Wu, Guanhua Wang, Siyuan Zhuang, Kehan Wang, Alexander Keimer, Ion Stoica, and Alexandre Bayen, "Composing MPC with LQR and Neural Networks for Efficient and Stable Control," *arXiv preprint arXiv:2112.07238*, 2021.

[183] Sepp Hochreiter and Jurgen Schmidhuber, "Long short-term memory," *Neural Computation 9(8)*, 1997.

[184] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[185] Nikko Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Interspeech*, 2015.

[186] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," in *MLSys*, 2021.

[187] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *SIGKDD (Tutorial Abstract)*, 2020.

[188] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu, "Pathways: Asynchronous distributed dataflow for ml," in *MLSys*, 2022.

[189] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Bentzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar, "A system for massively parallel hyperparameter tuning," in *MLSys*, 2020.

[190] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *SIGKDD*, 2019.

[191] Andrew Or, Haoyu Zhang, and Michael J. Freedman, "Resource elasticity in distributed deep learning," in *MLSys*, 2020.

[192] Chien-Chin Huang, Gu Jin, and Jinyang Li, "Swapadvisor: Push deep learning beyond the gpu memory limit via smart swapping," in *ACM ASPLOS*, 2020.

[193] Wikipedia, *Round-robin scheduling*, https://en.wikipedia.org/wiki/Round-robin_scheduling.

[194] Andrew Or, Haoyu Zhang, and Michael Freedman, "Virtualflow: Decoupling deep learning models from the underlying hardware," in *MLSys*, 2022.

[195] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis, "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines," in *MLSys*, 2022.

[196] Nvidia-occupancy, *Achieved Occupancy*, https://bit.ly/36S3bnG, 2020.

[197] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *USENIX OSDI*, 2020.

[198] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert Patton, "Fleet: Flexible efficient ensemble training for heterogeneous deep neural networks," in *MLSys*, 2020.

[199] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Zhi Li Pengyang Hou, Yihui Feng, Wei Lin, and Yangqing Jia, "Antman: Dynamic scaling on gpu clusters for deep learning," in *USENIX OSDI*, 2020.

[200] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *USENIX OSDI*, 2020.

[201] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi, "Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism," in *USENIX ATC*, 2020.

[202] Lukasz Wesolowski, Bilge Acun, Valentin Andrei, Adnan Aziz, Gisle Dankel, Christopher Gregg, Xiaoqiao Meng, Cyril Meurillon, Denis Sheahan, Lei Tian, Janet Yang, Peifeng Yu, and Kim Hazelwood, "Datacenter-scale analysis and optimization of gpu machine learning workloads," *IEEE Micro*, vol. 41, 2021.

[203] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park, "Elastic resource sharing for distributed deep learning," in *USENIX NSDI*, 2021.

[204] *CUDA multi-process Service*, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2021.

[205] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang, "Efficient rematerialization for deep networks," in *NeurIPS*, 2019.

[206] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," in *MLSys*, 2020.

[207] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock, "Dynamic tensor rematerialization," in *ICLR*, 2021.

[208] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *MLSys*, 2019.

[209] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *USENIX OSDI*, 2021.

[210] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers, "Pagoda: Fine-grained gpu resource virtualization for narrow tasks," in *Proceedings of Principles and Practice of Parallel Programming (ACM PPoPP)*, 2017.

[211] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *ACM ASPLOS*, 2013.

[212] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang, "G-net: Effective gpu sharing in nfv systems," in *USENIX NSDI*, 2018.

[213] Apple newsroom, *Apple unveils m1 ultra, the world's most powerful chip for a personal computer*, https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/, 2022.