# BUILDING AN EFFICIENT AND SCALABLE LEARNING SYSTEM ON HETEROGENEOUS CLUSTER

by

Donglin Yang

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2022

Approved by:

Dr. Dazhao Cheng

Dr. Weichao Wang

Dr. Dong Dai

Dr. Tao Han

# ABSTRACT

DONGLIN YANG. Building an Efficient and Scalable Learning System on Heterogeneous Cluster. (Under the direction of DR. DAZHAO CHENG)

Deep Learning (DL) has been widely applied in both academia and industry. System innovations can continue to squeeze more efficiency out of modern hardware. Existing systems such as TensorFlow, MXNet, and PyTorch have emerged to assist researchers in training their models on a large scale. However, obtaining performant execution for different DL jobs on heterogeneous hardware platforms is notoriously difficult. We found that current solutions show relatively low scalability and inefficiencies when training neural networks on heterogeneous clusters due to stragglers and low resource utilization. Furthermore, existing strategies either require significant engineering efforts in developing hardware-specific optimization methods or result in suboptimal parallelization performance. This thesis discusses our efforts to build an efficient and scalable deep learning system when training DL jobs in heterogeneous environments. The goal of a scalable learning system is to pursue a parallel computing framework with (1) efficient parameter synchronization approaches, (2) efficient resource management techniques, (3) scalable data and model parallelism in heterogeneous environments.

In this thesis, we implement robust synchronization, efficient resource provisioning approaches, asynchronous collective communication operators, which optimize the popular learning frameworks to achieve efficient and scalable DL training. First, to avoid the "long-tail effects" for parallel tasks, we design a decentralized, relaxed, and randomized sampling approach to implement partial AllReduce operation to synchronize DL models. Second, to improve GPU memory utilization, we implement an efficient GPU memory management scheme for training nonlinear DNNs by adopting graph analysis and exploiting the layered dependency structures. Third, to train wider

and deeper Deep Learning Recommendation Models (DLRMs) in heterogeneous environments, we propose an efficient collective communication operator to support hybrid embedding table placements on heterogeneous resources and a more fine-grained pipeline execution scheme to improve parallel training throughput by overlapping the communication with computation. We implement the proposed methods in several open-source learning frameworks and evaluate their performance in physical clusters with various practical DL benchmarks.

# DEDICATION

This thesis is dedicated to my parents for their constant support, encouragement, and belief in my dreams.

# ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Dr. Dazhao Cheng for his continuous support throughout my Ph.D. study. I cannot imagine having better advisors and mentors for my Ph.D. study. I would also like to thank my graduate committee members, Dr. Weichao Wang, Dr. Dong Dai, and Dr. Tao Han, for their help and encouragement throughout my Ph.D. study at UNCC. I also appreciate the camaraderie and support of my fellow lab member, Wei Rang. We spent an enjoyable time together brainstorming research ideas and tackling practical problems. I am grateful to my parents, Jihe Yang and Li Cai, for their support in pursuing my dreams.

The research and dissertation were supported in part by the US National Science Foundation research grants CCF-1908843 and CNS-2008265.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

## 1.1 Overview of Deep Learning Training in Heterogeneous Cluster

Deep learning (DL) [1] has achieved great success in various domains such as image classification [2], natural language processing [3], object detection [4], speech recognition [5], etc. Obtaining accurate deep learning models is a computation-intensive process, which requires large amounts of data and substantial computing capacity. Previous studies have shown that wider and deeper DNNs can significantly increase the model performance [6][7]. Recently, nonlinear architectures have been proposed to further improve the quality of image recognition tasks [8][9]. However, the limited size of GPU DRAM has been a major bottleneck for researchers to explore deeper and wider DNNs for better generalization performance. For example, it is reported that VGG-16 [10], which is composed of 16 computation-intensive convolution layers, requests a total of 28GB of memory usage for batch size 256 [11]. A representative nonlinear network, Inception-V4, requests up to 45GB memory to keep the entire network on the GPU in training [12]. However, the largest GPU memory capacity offered by the commercial NVIDIA Volta architecture so far is 32GB [13]. The memory shortage of GPU limits deep learning practitioners to deploy wider and deeper DNNs. There are many other system challenges for training deep neural networks.

The first part of this thesis focus on memory optimization for nonlinear networks. Many approaches have been proposed to reduce the GPU memory footprint of DNN training. However, these solutions have their limitations. For example, most prior works propose reducing the model size to reduce the memory footprint. However, this strategy either provides low memory footprint reduction or results in a loss in training accuracy [14][15]. Firstly, in DNN training, parameter weights only account for a

small fraction of the total memory footprint. In training, intermediate feature maps are the primary contributor to the significant increase in memory footprint in DNN training. These intermediate values should be stored/stashed in the forward pass so that they can be reused later in the backward pass. Additionally, approaches that apply lower precision computations for DNN training, mostly in the context of ASICs and FPGAs, either do not target feature maps (and thus achieve low memory footprint reduction) or result in reduced training accuracy [16]. Memory compression [17] and data encoding [18] is another approach to reduce the GPU memory requirement for training, which, however, introduces high-performance overhead. State-of-the-art memory footprint reduction approaches for training swap data structures in and out between host and device memory [19][20]. However, existing swapping approaches are inefficient in reducing the memory footprint for training.

Inspired by the fact that DNN training follows a series of layer-wise computations, vDNN [19] and SuperNeurons [20] propose to virtualize the memory usage of deep neural networks across both GPU and CPU memories. Considering that GPU can only process one layer at any given time, it is not necessary to overprovision the memory allocation to accommodate the entire neural network on the GPU. vDNN and SuperNeurons release or move data structures, particularly the intermediate feature maps, between CPU and GPU, by exploiting the inter-layer dependencies and reuse patterns of DNNs. However, those techniques are not well-tuned to address the dependency and memory variations in nonlinear networks. Firstly, the core idea of vDNN and SuperNeurons is to offload data of one network layer when it is not required in the near future and can be released from GPU DRAM, saving space for other layers. The offloaded data is brought back to the GPU when needed in the backward pass. It can achieve optimal performance if the communication between CPU and GPU can be well hidden by computation to utilize the bandwidth. However, we observe that offloading data structures from the GPU to CPU or prefetching data back to GPU

from CPU layer by layer brings significant inefficiency. For example, transfer time can be longer or shorter than the forward computation time across layers so that the communication can only be partially overlapped with the computation. Usually, the communication time is much longer than the computation time in Pooling layers. In contrast, the computation time is usually much longer than the communication time in convolution layers. Specifically, for nonlinear blocks, where there are join or fork connections, more benefit can be earned by aggressively advancing the computation in the forward pass or the prefetching operations in the backward pass. Secondly, none of the existing work presents an efficient solution to handle the memory fragmentation problem for nonlinear networks. Different from linear networks, which follow a simple and fixed execution pattern, causing negligible memory wastage, nonlinear networks exhibit varied dependencies and dynamic references. As a result, those complex nonlinear blocks, which has different data size, varied resident duration, and dynamic reference counts, interleave with layers that have simple dependencies.

The first work demonstrates that the default memory management can lead to higher fragmentation because the released memory regions cannot be coalesced into a larger one, resulting in free but usable space. Overall, this work proposes and designs Dymem [21], a novel approach for training nonlinear networks. Instead of using a layer-by-layer strategy, Dymem adopts a more greedy asynchronous solution to maximize the DRAM bandwidth, balancing memory usage and performance improvement. Furthermore, this project first analyzes the root cause of GPU memory fragmentation in DNN training. Then, a Group Tensors By Mobility (GTBM) placement policy is designed to allocate tensors on the proposed unified memory pool based on mobility, exploiting the dependencies and reuse distances.

The second part of this thesis introduces a new solution to mitigate stragglers for distributed training. There is a trend to distribute the training process across clusters to accelerate the training process. Distributed training is an iterative process, which

adopts the most popular algorithm called mini-batch Stochastic Gradient Descent (SGD) [22] to compute gradients and update models until convergence. Data parallelism [23][24][25] is one of the most popular approaches to distributing the training process. In this strategy, different machines in a distributed environment have a complete copy of the model. The procedure of distributed training is bottlenecked by the parameters communication. In a cluster environment, two typical approaches are widely used to synchronize model parameters at the end of each iteration: centralized [23] and decentralized algorithms [26]. For the centralized approach, nodes are divided into two categories: Parameter Servers (PS) and workers. PS stores the model parameters while workers execute the training process in each iteration. In the decentralized approach, every worker performs the computation and maintains a copy of the parameters. Recently it has been theoretically proven that decentralized approaches can outperform centralized ones [26].

In a distributed fashion, the main performance bottleneck comes from the communication hotspot, which is caused by the frequent access to global models. At the end of each iteration, gradients or parameters are frequently transferred among workers until the model parameters are fully updated. In PS, all nodes have to communicate with central servers, leading to a communication bottleneck. It is reported that more than 90% of iteration time is required for communication for a wide and deep neural network model, e.g., VGG16 [27]. This problem has been alleviated with decentralized algorithms, which implement all-to-all communication logically. This bandwidth-optimal communication protocol has been shown to outperform centralized approaches, especially for neural networks with large models. Moreover, decentralized algorithms can achieve better scalability, which is independent of the number of workers. In this thesis, we focus on one of the most popular implementations of decentralized algorithms, Ring AllReduce [28]. The execution of AllReduce follows the Bulk Synchronous Parallel (BSP) model. In it, parallel processes execute the

same task at the same iteration, and the generated updates must be synchronized on parameters when all tasks are finished. The strict global barrier at each iteration ensures the model's accuracy but makes it vulnerable to "long-tail effects." All processes have to wait for each other to complete the propagation before the AllReduce operation is triggered. The slowest worker bounds the performance. Recently, though many studies have been proposed to overcome stragglers in the centralized fashion [27][29][29][30], it falls short in supporting decentralized approaches.

This work investigates the causes of straggler in terms of computation. Firstly, the straggler can be caused by the inherent load imbalance. A dynamic neural network such as LSTM and RNN [31] model sequences of data (e.g., video and sentences). The computation graph topology depends on input values, whose data samples could have variable shapes. For example, the recurrent structure of the network leads to the training overhead being proportional to the length of the input video. Also, the sentences in the training dataset for a language model [32] have various input lengths, resulting in an unbalanced workload across different batches. We observe that varying input lengths result in computation load imbalance, leading to inefficiency during the synchronization phase. Secondly, the heterogeneity from the system itself can also cause "long-tail effects." Shared clusters and clouds often exhibit significant hardware and performance heterogeneity due to multi-tenant interference and continuous machine maintenance [33][29].

Several pieces of research have been proposed to explore the robustness of deep learning processes [34][35]. In particular, AD-PSGD [34] is the first that proposes to use randomized communication to reduce the effects of stragglers probabilistically. In contrast to waiting for all processes, each worker randomly selects one worker to average parameters between the two. However, this design incurs significant synchronization overhead to ensure atomicity. It also requires manual efforts to avoid scheduling conflicts[35]. Inspired by the fact that the deep learning training process

is robust to bounded errors, we propose to relax the global barrier without changing the communication graph to mitigate the impact from "long-tail effects." This thesis offers a new solution, called Randomized Non-blocking AllReduce [36], that allows the AllReduce operations to proceed with the synchronization without waiting for the completion of all processes' computation.

The challenge of partial AllReduce lies in when and how to terminate input data processing. For instance, each process is unaware of the progress of others in the existing computing platforms. The implementation of Horovod requires when all of the processes inform gradients' readiness, and then the synchronization is executed. In a global view, the critical challenge to achieve a partial AllReduce is deciding the time to trigger the sync, i.e., determining the number of processes contributing to their updates. It is a trade-off between system efficiency and algorithm efficiency. To tackle the challenge, this work adopts the power of two choices load balancing technique [37] to partial schedule synchronization by probing two random processes and determining the synchronization time based on the faster one. In a local view, this work uses two individual threads to execute computation and communication, through which cross-iteration training is enabled. In a heterogeneous cluster, the deterministic performance difference between machines can not be neglected. To utilize the flexibility of the traditional PS to achieve asynchronous updates, this work combines PS with AllReduce architecture to implement a hierarchical synchronization protocol. The convergence analysis shows that the error is bounded, and the statistical properties guarantee the convergence of deep neural network models.

The third part of this thesis introduces a new system design to train wider and deeper sparse models. Recommendation models are widely used across a variety of Internet services such as personalized advertisements, entertainment, e-commerce, and search [38][39][40]. Recommender systems can help companies such as Amazon and Facebook retain customers by providing tailored suggestions specific to their

needs. It is reported that the training workloads for deep learning recommendation models account for more than 50% of the total AI computing cycle at Facebook [41].

Different from classic machine learning models or deep learning models used in natural language processing (NLP) and computer vision (CV), deep learning models for recommendation systems uses a bunch of sparse categorical features. Sparse features have mostly zero values and have extremely high dimensions. For this reason, deep learning recommendation models are also called sparse models. To achieve better prediction accuracy, models size for recommendation systems grows into multi-terabytes, and the size of sparse features could be up to millions, even billions, leading to the fact that it is difficult to host the wide and deep models on a single device. Most of the existing work choose to place the large embedding table on host memory because of the larger capacity [42][43]. GPUs attract broad attention as deep learning accelerators. However, it is difficult to directly employ GPUs on training sparse models considering the limited capacity of the device and the large volumes of data transferred over PCIe. Prior efforts [24] cache the large embedding table on host memory and update parameters on the device, which still introduces extremely high overhead on PCIe due to the communication between CPU and GPU. Facebook prototypes Zion GPU server equipped with 8 GPU accelerator, which provides the high compute and memory capacity. However, due to the lack of the direct interconnect among devices when placing the embedding tables on GPUs, the Zion GPU server cannot achieve the optimal training performance [41]. This paper will further discuss the challenges of using GPU parameter servers in the heterogeneous cluster and propose several techniques to achieve system scalability.

PaddlePaddle [44] has investigated the initial design of the hierarchical parameter server to combine GPU and CPU to host the large recommendation models. There are still some key points that could be potentially leveraged for further performance improvement. In PaddlePaddle, Remote Direct Memory Access (RDMA)

communication protocols are adopted to support direct point-to-point devices communications but do not need to involve CPUs. However, the commonly used library such as NVIDIA NCCL [45] only provides GPU-GPU communication. A mixed embedding table placement requires CPU-GPU data transfer first, which introduces overhead on PCIe and causes additional delay. Furthermore, the provided collective communication primitives are blocking and synchronous, which could cause network contention and deteriorate the end-to-end performance. Second, due to the expensive cross-server Alltoall and Allreduce operations, the communication overheads are not negligible even on servers where GPUs within the server are connected by dedicated interconnects like NVLink. Pipeline parallelism is an efficient strategy to overlap inter-node communication with computation. Without a carefully designed scheme, excessive data transfer can be a dominant performance slowdown. The third challenge is to place embedding tables on GPUs and CPUs effectively. However, modern deep learning accelerators have larger device memory capacities. The device memory is still very precious, considering that the size of embedding features has grown dramatically from tens of gigabytes to terabytes in the industry recently. How to partition embedding tables across GPUs and CPUs should be carefully determined.

In this thesis, the major contribution is to design and implement an efficient and scalable training system for wider and deep neural networks in the heterogeneous cluster. In a nutshell, high-performance components and policies can be implemented to achieve better training throughput and higher resource utilization. The contributions of this thesis can be summarized as (1) An efficient GPU memory management runtime for training nonlinear neural networks to support wider and deeper neural networks. (2) a scalable and asynchronous AllReduce scheme to mitigate the impact of stragglers caused by load imbalance. (3) a high-performance collective operator to support hybrid embedding tables placements and a more fine-grained pipelining scheme to improve the end-to-end training throughput.

# CHAPTER 2: Background and Motivation

## 2.1 Addressing GPU Memory Shortage for DNNs Training

DNNs typically composes of an input and output layer with multiple hidden layers in between. Due to the limited GPU memory capacity, efficient memory management is important to run deeper and wider DNNs on GPU. This section summarizes the DNN models, current memory management optimization technique and motivates our work to overcome existing limitations.

### 2.1.1 DNNs Training on GPUs

DNN training is based on a set of inputs and obtained outputs. The training process consists of two phases: forward and backward passes [46][47]. Specifically, the backward propagation algorithm is to propagate the error and search the gradient of the loss function that can be applied to adjust the parameters towards improving accuracy [48]. It consists of four types of data structure in the DNN training: feature maps, weights, gradients, and workspace. Feature maps are the intermediate results that are consumed in the following forward or backward layers. Gradient maps are

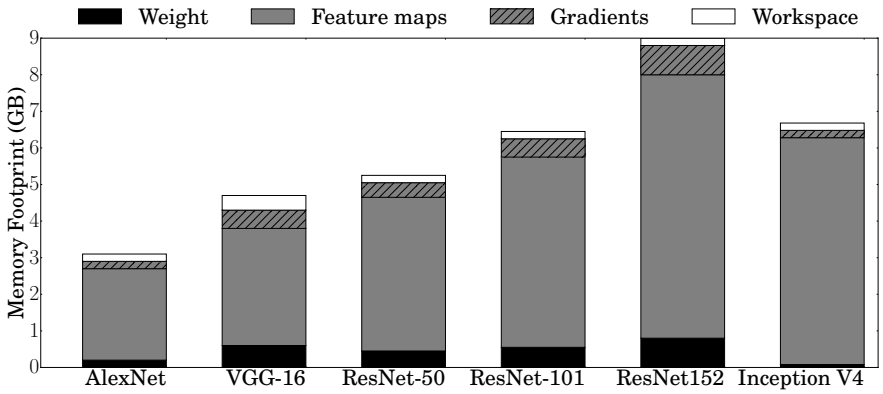
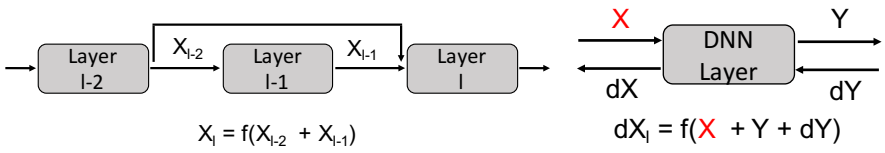



Figure 2.1: The breakdown of memory footprint in DNN training for different networks (GB).




(a) Forward pass. (b) Backward pass.

Figure 2.2: Forward and backward computation. $X$, $Y$, $dX$ and $dY$ are input feature maps, output feature maps, output gradient maps and input gradient maps respectively.

the intermediate results that are generated in the backward pass and consumed by the dependent layers. Weight value decides how much influence the input will have on the output. The workspace is the intra-layer storage to speed up the layer computation. In particular, the workspace requires additional but temporary GPU DRAM to achieve better performance of the Convolution algorithm. In the forward pass, the input feature maps $X$ are fed to the current layer in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function, and passes its output to the successive layer. Nonlinear networks contain one-to-many (fork) and many-to-one (join) inter-layer dependencies. For example, feature maps $X$ from layer $l-2$ and layer $l-1$ $l$ are joined as the input for layer $l$, as shown in Figure 2.2a. Backward propagation computes the gradient in the weight space of a feedforward neural network, with respect to a loss function. Typically, in a backward calculation, a layer requires it stashed input feature maps $X$, output feature maps $Y$, and input gradient maps $dY$ to obtain the output gradient maps $dX$, which can be shown in Figure 2.2b. The backward propagation can only be performed when all of these required data structures are available on GPU.

Traditional Convolution Neural Networks (CNN) typically consists of several basic building layers, including Convolution (CONV), Pooling (POOL), Activation (ACT), Softmax, Fully Connected (FC), Batch Normalization (BN), and Dropout layers. A linear neural network is structured as a sequence of independent and inter-

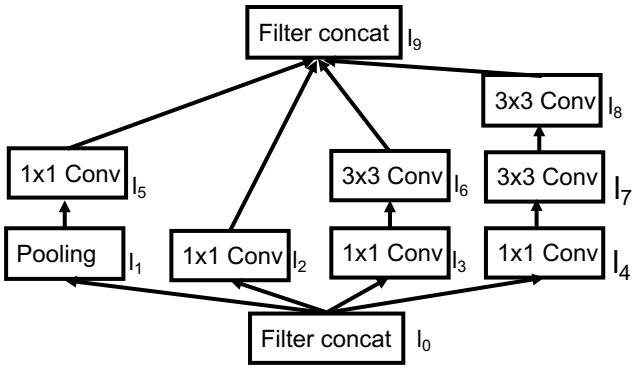
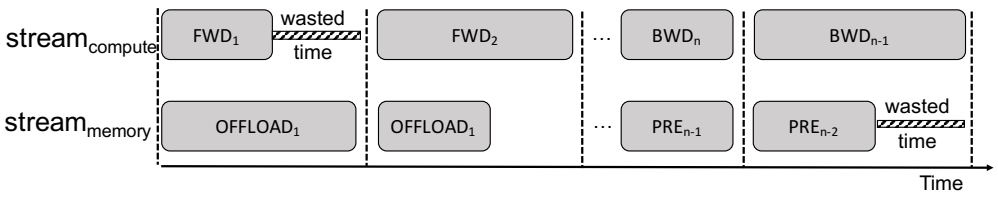



Figure 2.3: Schema of Inception-v4 network.

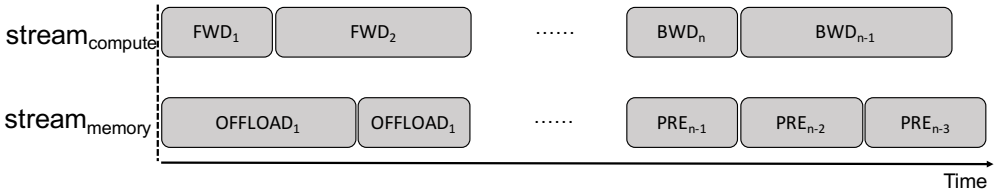
connected instances. Recently, several nonlinear networks, such as Inception V4 [8] and ResNet [49], have been proposed to improve the state-of-the-art performance of image classification further. Training the elaborate neural network exhibits significant challenges considering the limited GPU memory. To understand the memory consumption behavior of DNN training, we conduct a study on GeForce GTX TITAN X GPU with six representative CNNs. Figure 2.1 shows the breakdown of the GPU memory footprint. It can be learned that the GPU memory tends to be mostly occupied by feature maps. For example, more than 90% of GPU DRAM is required by Inception V4. Furthermore, it is a waste to reside those feature maps on GPU memory even though they have future but distant dependencies, especially for the very deep neural networks. Thus, the intermediate feature maps are the key factors for optimizing memory usage in DNN training.

### 2.1.2 Nonlinearities in DNNs

For linear networks, data is sequentially propagated in both the forward and backward passes, following a fixed sequential execution pattern. Compared with linear networks, nonlinear networks have a high degree of dependency variations. To illustrate these, we use a representative nonlinear block from Inception-V4 [8] as an example, which is shown in Figure 2.3. There are two simple nonlinear connections: fan and join. In this example, the fan connection creates four branches after layer $l_0$.




(a) Default approach.




(b) Aggressive approach.

Figure 2.4: Synchronization w/i and w/o barrier.

Each of them has a different number of layers. Each branch has to finish its computation before it reaches the join connection, i.e., layer $l_9$. Nowadays, a deep nonlinear network could have hundreds of fan and join connections inside the network, resulting in a complex network architecture [50]. Note that the GPU can only process a single layer's computation at any given time due to such inter-layer data dependencies. In terms of memory allocation, care should be taken because, in nonlinear networks, multiple layers consume the output feature maps from a previously processed layer in a fan connection. For example, the output feature maps from layer $l_0$ can only be released from GPU memory until layers $l_1$, $l_2$, $l_3$, and $l_4$ have been propagated. Similarly, in the join connection, all the output feature maps from preceding layers should reside on the GPU until their final consumer has completed the propagation. These nonlinear variations complicate runtime resource management, which requires a more efficient solution.

### 2.1.3 Motivation for Efficient GPU Memory Management

#### 2.1.3.1 Memory Offload/Prefetch for DNNs

Several memory-reduction techniques have been proposed to address the problem of limited GPU resident memory. For example, vDNN [19] and Superneurons [20] choose to offload selected layers to the preallocated pinned CPU memory and prefetch these data back to GPU when required. Typically, in the forward pass, the input feature maps $X$ from the preceding layer can be offloaded to CPU memory if there is no more dependency, after which these data can be released from the GPU memory. The runtime uses two independent processes to complete the computation and communication, which enables the CPU-to-GPU data transfers to overlap with the computation asynchronously. In the backward pass, for those offloaded layers, the runtime should bring the feature tensors $X$ back to the GPU DRAM before the backward dependent layer starts its propagation. The prefetch operation for layer $m$ can be overlapped with the computation of layer $l$ in the backward pass, where $l > m$. Ideally, this design can maximize the performance by hiding the communication by the computation time. To ensure the safety of parallel streams, they enforce a synchronization at the end of each layer, which means the communication and computation stream cannot advance each other in both the backward and forward passes, as shown in Figure 2.4a. However, this design largely depends on the communication/computation ratio. It works well for the linear network, e.g., VGG-16 [10], which consists of twelve computation-intensive convolution layers. The offload/prefetch operation can be well hidden because convolution layers require longer computation time than transfer. This is inefficient, particularly for nonlinear networks. To demonstrate this, Figure 2.5 presents the communication and computation time for ten layers in both the forward and backward pass of GoogleNet [8]. From the Figure, $f_5$'s computation time is much lower compared to offloading time, while the next layer's forward computation time is higher than the communication time. If $f_5$'s input is decided to

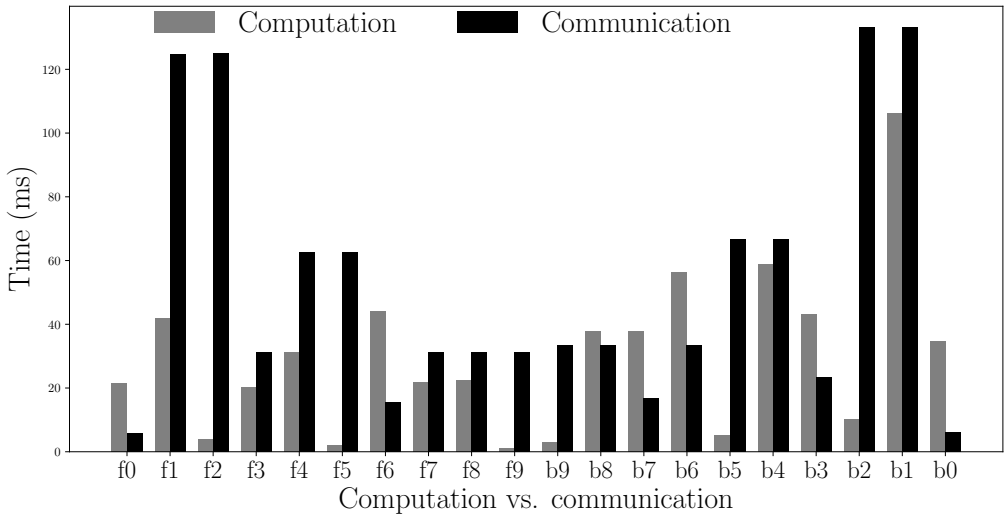



Figure 2.5: Computation and communication time for different layers in forward and backward pass.

be offloaded, then it is not necessary to wait for the offloading of $f_5$ before starting the next layer's computation. A more efficient synchronization without a barrier is shown in Figure 2.4. Similarly, in the backward pass, when the layer $b_7$ is being propagated, the prefetching operation for layer $b_5$ can be initiated after the transfer of layer $l_6$ is finished. Secondly, the backward pass of each layer requires memory space for gradients input and output maps besides input and output feature maps compared with forward. Hence, in the forward pass, its peak memory requirement is not higher than the backward pass if this aggressive strategy is adopted to advance computation. However, care should be taken in the backward pass because aggressively prefetching data does not always bring the benefit. These observations motivate us to propose a more efficient memory scheduling strategy to balance memory saving and performance.

#### 2.1.3.2 GPU memory fragmentation

Training a deep DNN on GPU with limited memory results in frequently caching and freeing tensors in a training iteration. To avoid the nontrivial allocation and deallocation overhead from using the native CUDA API, *cudaMalloc*, and *cudaFree*,

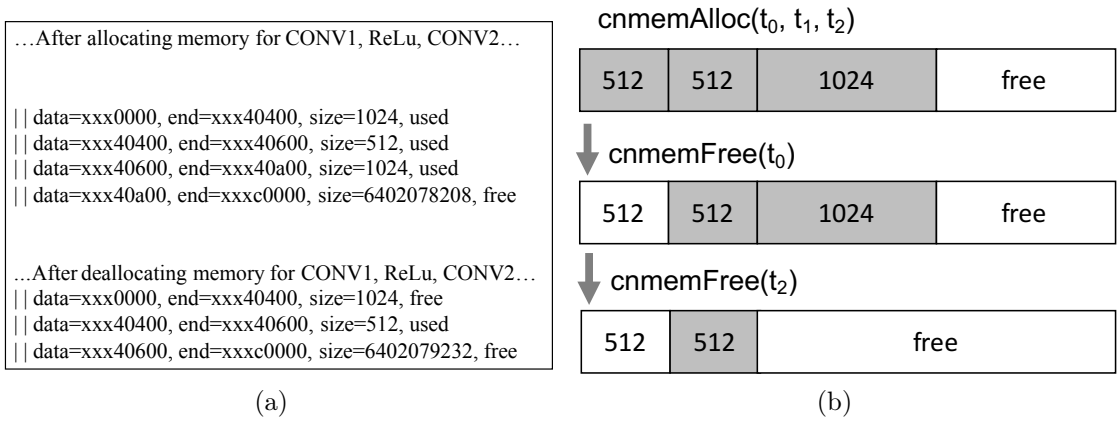
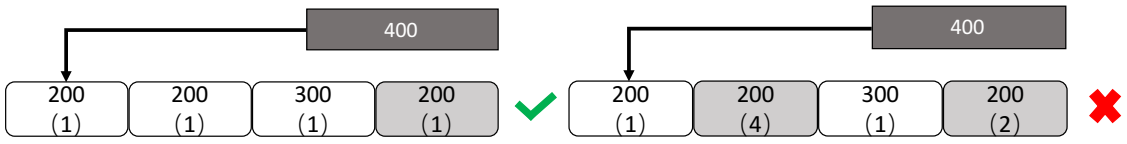



Figure 2.6: GPU memory allocation process.

GPU memory pool is always adopted as an effective memory optimization technique [20][19]. It preallocates a continuous chunk of memory as a shared memory pool and takes over memory management from the operating system. The preallocated memory pool returns a list of allocated but empty addresses. For an allocation request, the memory pool finds the first node with enough free memory from the empty list. After that, it updates the available list and the occupied list to track memory usage. For a deallocation request, the memory pool locates the node in the allocated list with the hash table, and then the pool puts the node back to the empty list. The sequences of allocation/free do not affect training but can impact the amount of fragmentation. For example, memory for the input of the layer to be prefetched could be allocated before/after allocating space for gradients of the input of the layer to be propagated in the backward pass. Figure 2.6 demonstrates the allocation process of CNMeM [51], which is a GPU memory pool developed by Nvidia. We allocate three tensors on the GPU, which require 512, 512, and 1024MB GPU memory, respectively. Then tensor $t_0$ and $t_2$ are freed from GPU. From the log information 2.6a, we can see that coalescing operations can combine available contiguous regions into a lager page in the same virtual space. However, the released region for $t_0$ cannot coalesce with the other available regions because they are not in




(a) Linear network. (b) Nonlinear network.

Figure 2.7: 400 represents the memory request. (4) represents the reference counts. White blocks represent free regions, which can be coalesced if they are contiguous while grey blocks represent occupied regions.

the continuous address, which is illustrated in Figure 2.6b. This buffering and paging strategies work at the coarse memory granularity, which results in inefficiencies for nonlinear networks whose data sizes and dependencies vary significantly.

**Case study:** Figures 2.7 represent two simple examples of linear and nonlinear networks. In a linear network, all layers have only one reference. If a coming allocation request for tensor is 400MB, for the linear network, it can serve the request because the coalesced regions from the released tensors have enough space. However, for the nonlinear network, though there is 200MB and 300MB empty list, it can not serve the incoming request because they are distributed at two separate lists. Between these two fragmented spaces, the allocated tensor resides on the GPU longer than other tensors because multiple layers have dependencies on it. Fragmentation happens at the virtual address level. It is not allowed to modify the page tables on the GPU. And it is also impossible to move data around the GPU with negligible overhead. The best-fit algorithm requests an additional 400MB memory to satisfy the demand. When peak memory consumption is close to the GPU memory capacity, this fragmentation might impact the trainability of networks.

Furthermore, from Figure 2.8, we can learn that the memory usage of CONV, ACT, BN, and POOL layers can account for more than 90% of total usage. However, it is not fruitful to offload Dropout, Softmax, and FC layers because they only require less than 1% of the total memory. These layers require less memory but stay longer until no more dependency. It requires us to manage the allocation of these tensors carefully.

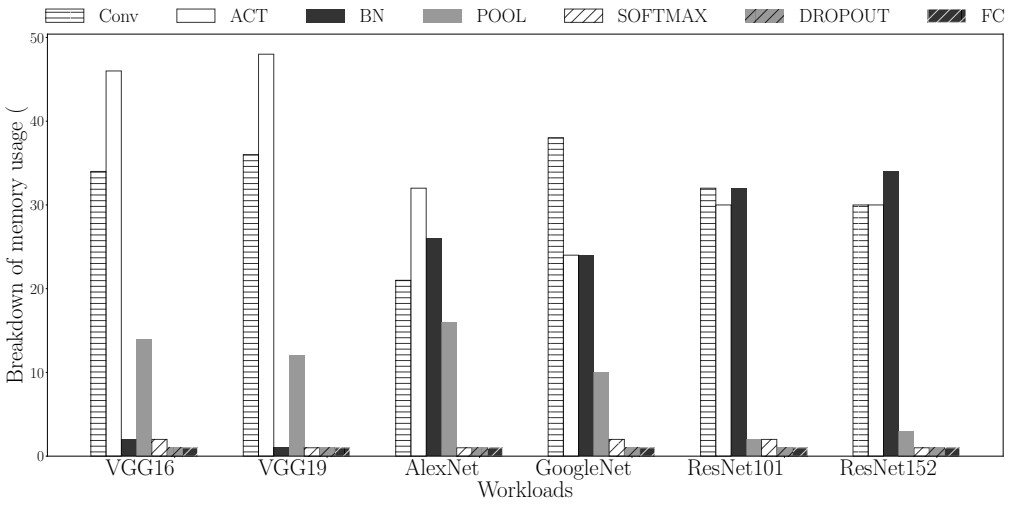



Figure 2.8: The fraction of memory usage by various layers for different networks.

Otherwise, it can cause further fragmentation. Based on the above observations, an efficient tensor placement policy should be proposed while considering both the varied data sizes and graph dependencies.

## 2.2 Mitigating Stragglers for Distributed Training

### 2.2.1 Distributed Deep Learning

In the training stage of deep learning, Stochastic Gradient Descent (SGD) [52] is adopted to minimize the loss function $f(x)$ over a data set $S$. In each iteration, parameters $x$ are updated by $x \leftarrow x$ - $\gamma \cdot \nabla_x f(x;\xi)$, where $\gamma$ represents learning rate, and $\xi$ represents a mini-batch of randomly sampled data from $S$. With the growing volume of data, it is popular to parallelize the training process in a distributed environment. Multiple parallelism schemes have been proposed recently to distribute the training process: data parallelism [28][23], model parallelism [53], hybrid parallelism [54][55] and pipeline parallelism [56][57]. Among them, data parallelism is the easiest one to be implemented without significant statistical efficiency loss compared with other approaches. Therefore, many popular deep learning frameworks such as TensorFlow [58], PyTorch [59] and MXNet [60] support this approach. Our paper focus on the data parallelism model.

For the data parallelism model, each node processes the randomly sampled input data independently and obtains gradients using the backpropagation algorithm [48]. At the end of each iteration, the obtained gradients from each node need to be gathered around so as to update the global parameter during the synchronization phase. The updated model will be applied in the next iteration and the distributed training process keeps this procedure until the model convergences. Synchronization is an essential part of parallelizing the training process and plays a critical role in achieving better scalability in a heterogeneous environment.

### 2.2.2 Existing Synchronization Approaches

Parameter Servers (PS) is a well-known scheme for parallel SGD execution, which is also called as the centralized algorithm. Parameter Servers and multiple workers are launched in this typical setting. At each iteration, each independent worker obtains gradients based on the SGD and sends the data to the central PS. The central PS will send back the updated model to each work and continue the training process. However, this simple approach has a significant drawback because all the workers have to push/pull parameters from the centralized servers, leading to the communication hotspot. The communication hotspot limits the system scalability. Decentralized training is proposed to alleviate this issue.

**Algorithm 1** The decentralized training process.

**Require:** A set of workers $M$; the communication topology $G$.

1: **for** worker $m_i \in M$ **do**
2: compute the gradient $g_{k,i} = \nabla_{\varepsilon_{k,i}} f(x_{k,i}; \xi_{k,i})$
3: average gradients using AllReduce $\overline{g}_k \leftarrow \sum_{i \in M} g_{k,i}$
4: update parameters $x_{k+1,i} \leftarrow x_{k,i}$ - $\gamma_k \cdot \overline{g}_k$
5: **end for**

As shown in Algorithm 1, in a decentralized setting, there are no central parameter servers. Every worker in this scheme maintains a complete copy of the model parameters. At the end of an iteration, each node sends the obtained gradients to

their out-going neighbors according to the communication topology, after which it applies the obtained gradients to the parameter. Ring All-Reduce [28] is one of the most popular implementation, which works in a scatter-and-gather way. In this setting, each worker only communicates with its neighboring sender and receiver in a fixed order during the synchronization phase, forming a logical ring. For a distributed system with $N$ servers, in each step, the worker sends one portion of $\frac{1}{N}$ gradients to its left neighbor and meanwhile, it accepts another $\frac{1}{N}$ gradients from its right neighbor. It then averages the accepted gradients with its local portion, which is called as *Reduce* operations. The averaged portion will be transferred to its left neighbor in the next step during the scatter operations, and meanwhile this worker will accept another averaged part from its right neighbor. After $N-1$ steps, scatter operations are finished, and every worker owns a complete $\frac{1}{N}$ of gradients. Gather operations will be triggered then, which works similarly, but it does not require *Reduce* operation. In each step, each worker replaces its local portion of gradients with the accepted one from its right neighbor. After $N-1$ steps, each worker obtains the whole set of global gradients. It benefits from contention-free communication compared with PS strategy by abandoning the many-to-one communication protocol, which achieves the ideal parallelism within the theoretical upper bound. These procedures guarantee consistent convergence with the expense of introducing a blocking barrier. Workers will always have to wait for the slowest one to finish at each iteration. This means that this distinct communication pattern can only achieve its best performance in a **homogeneous** environment. This is a strict requirement, especially in a shared cloud environment.

To tolerate system heterogeneity, AD-PSGD [34] proposes a random synchronization mechanism to enable the point-to-point communication. Instead of synchronizing with the fixed neighbors specified by the communication topology, a worker performs an atomic model averaging with the randomly selected neighbor, regardless of whether

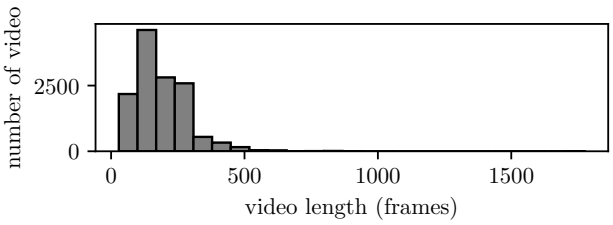
they are in the same iteration or not. Even though the slow workers inevitably have staler parameters, the effects on the training of the global model can be minimized via the probabilistic solution. However, this strategy requires additional system overhead and manual efforts to generate a dynamic communication graph [35].

### 2.2.3 Challenges and Motivation

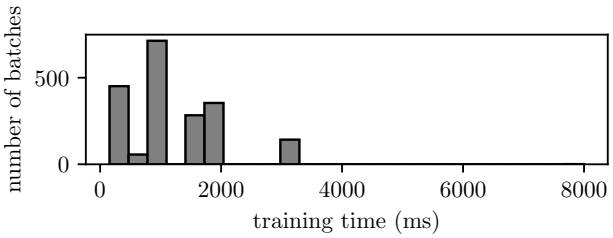
#### 2.2.3.1 Case Study

In a shared cluster, major bottlenecks for distributed deep learning training comes from the straggler problem, which is caused by various heterogeneities. In general, deterministic heterogeneity is relatively common because a large cluster is always configured with different hardware and dynamic capacities. Moreover, DL workloads running in a shared cluster always coexist with other data analytic workloads, which introduces transient slowdowns. The over-subscription of workloads in a cluster further harms the performance of DL training. The strict requirement for AllReduce operation degrades the system efficiency, slowing down the training progress. System heterogeneity is common, especially in the cloud environment. The straggler issue will be more severe on a large scale.

Load imbalance from the application also widely exists in the training of deep learning models. We use Inception V3 [8] to extract video features for UCF101 [61], which has 13,320 videos. Figure 2.9a summarizes the distribution of the length of video frames. The lengths of video range from 29 to 1776, with a mean value of 186 and a standard deviation of 97.7. We run a Long Short-Term Memory (LSTM) [62] model to demonstrate our observation. We configure the batch size as 32 and train the model on the GPUs with the same capability. Figures 2.9b illustrates the training time distribution over the 2,000 sampled batches in two epochs to train a 2,048-wide single-layer LSTM model on video frame features. Due to the network's recurrent structure, the computational overhead is proportional to the number of frames in the input video. The training time is distributed from 156 ms to 8000 ms, with a mean runtime of 1,219




(a) Video length distribution.



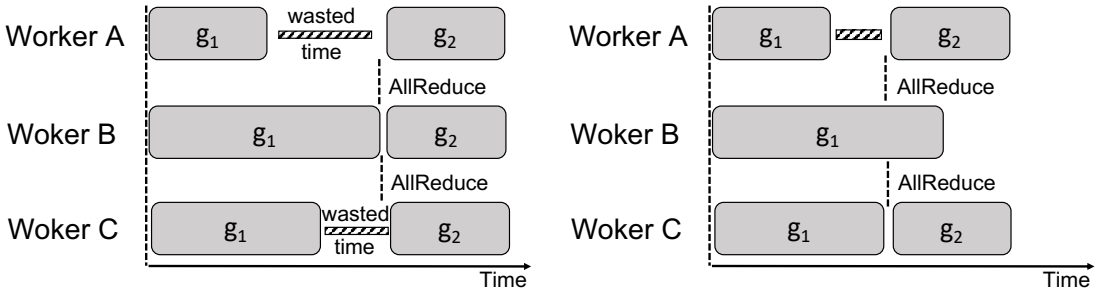

(b) Training time distribution.

Figure 2.9: Inherent load imbalance from training LSTM on UCF101.

ms and a standard deviation of 760 ms. These statistics above show that training an LSTM model for video classification exhibits an inherent load imbalance. Load imbalance is common, especially for dynamic neural networks [63]. Since sequences may have variable length, the cell function is executed for a different number of times for different sequences. In dynamic neural networks, some dependencies depend on input data or parameter values, which is dynamically determined, resulting in the runtime imbalance.

#### 2.2.3.2 Non-Blocking Reduce Synchronization

The decentralized training follows the Bulk Synchronous Parallel (BSP) model, in which workers synchronize at the end of an iteration, i.e., barrier and proceed after the model parameters have been fully updated by all workers. However, from the above observations, we can learn that this synchronous execution lowers hardware efficiency since fast workers have to wait for stragglers to complete each iteration, wasting computing cycles, which can be illustrated in Figure 2.10a. Bounded staleness [64] is an important technique to tolerate the temporary slowdown for centralized algorithms,




(a) default AllReduce. (b) Non-blocking AllReduce.

Figure 2.10: $g_1$ and $g_2$ represent gradients from iteration $t_1$ and $t_2$, respectively. The length of the bars represents the training time. Worker $C$ is specified as the initiator.

allowing faster workers to advance to the next iteration within the bounded staleness. It is easy to implement asynchronous synchronization in a centralized scheme because each worker communicates with parameter servers directly and computes gradients independently. However, the distinct communication pattern of the Ring All-Reduce protocol makes it difficult to enforce such a technique directly. Because current implementations require that all the results should be within the same iteration to ensure consistency.

These observations motivate us to propose a Non-Blocking-Reduce mechanism to synchronize partial results without changing the communication graph. Inspired by the fact that the training process is robust concerning bounded errors, we propose to update the gradients without waiting for slower processes to reduce the delay. Instead of waiting for the slower nodes, the faster worker will Reduce the gradients partially from available workers. Figure 2.10 shows a simple example in a decentralized setting with three workers. Worker $A$, $B$, and $C$ are at the iteration $t_1$ in the beginning. In default, when worker $A$ completes the propagation, since the worker $B$ and $C$ are still in progress, it has to wait for the completion of the other two processes. It is until the slowest process, i.e., $B$, finishes the propagation, then the AllReduce is executed. The default strategy under-utilizes the resource because faster processes keep idle when waiting for slower processes. In a non-blocking setting, an initiator is randomly

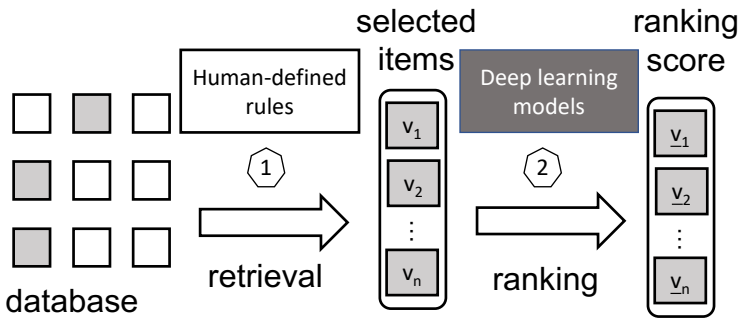



Figure 2.11: Typical model architecture.

selected among these three processes. Suppose that $C$ is selected. When process $C$ finish propagation, the AllReduce operation is enforced, then worker $A$ and $C$ can advance to a new iteration. During the synchronization phase, worker B contributes a $Null$ value to maintain the communication graph. In this way, the waiting time for $A$ is reduced while there is no idle time for $C$. The overall system efficiency is improved compared with the default strategy. In the next iteration $t_2$, if both worker A, B and C have available results, the AllReduce operation will synchronize the gradient $g_1$ from worker $B$ with $g_2$ from worker $A$ and $C$ though they are in different versions. Through the Non-Blocking-Reduce mechanism, the strict blocking barrier can be relaxed, which can reduce the impact of stragglers. In a cluster with $P$ processes, the probability that any process is specified as the initiator is equal to $\frac{1}{P}$, correspondingly on average, 50% of processes join the collective operation. In this way, the asynchronous synchronization might lower statistical efficiency than the synchronous implementation but it can trade statistical efficiency for system efficiency. However, the above example is too simple and straightforward with only two workers. In the following sections, we will discuss how to extend the Non-Blocking-Reduce mechanism to a cluster-wide scale and heterogeneous environments.

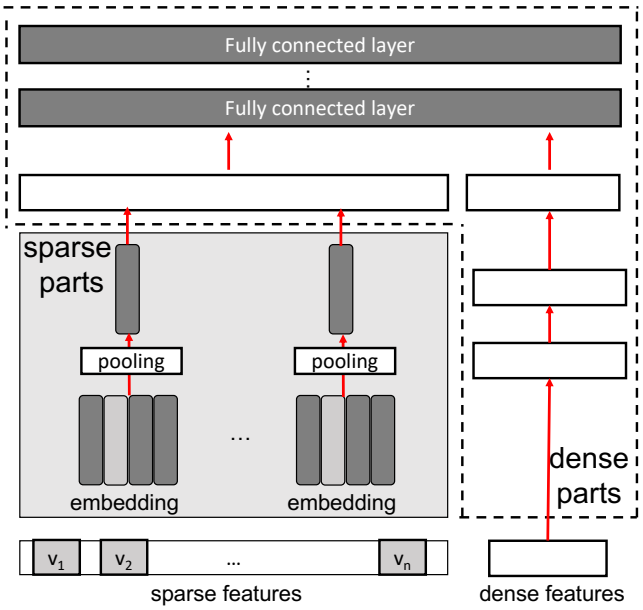



Figure 2.12: Deep learning recommendation model.

## 2.3 Training Wider and Deeper Models

### 2.3.1 Parallel Paradigm of Recommendation Models

Figure 2.11 presents the typical workflow of the recommendation model. Given a request query for user $u$ and item, the recommendation system follows a two-step process to predict a preferred item $v_i$, which is selected from a huge database whose size could be up to hundreds of billions. The first step is called retrieval, which reduces the size of the candidate pool to hundreds of items by applying the human-defines rules. The next step is called ranking, which adopts deep learning models to generate the ranking scores for selected items. The predicted result is the ranking score of a user action $h$, (e.g., click, favorite) when the user $u$ clicks on an item $t_i$, which is defined as $p(h|u, t_i)$. To provide an accurate prediction result, DLRMs utilize abundant sparse data and complicated deep neural network models [65]. Sparse categorical features are adopted to interpret the interactions between two items, which have mostly zero values. To efficiently represent sparse features, DLRMs often adopt a technique called sparse embedding, which transforms the sparse data into a more compact and low-

dimensional dense format, which is illustrated in Figure 2.12. A sparse data is mostly represented in the form of one-hot encoding. The transformed data are often called as the sparse part for the deep learning recommendation models. During the sparse forward pass, a specific feature ID is coupled with a related embedding table, and each encoded feature will be used to lookup a distinctive column. The lookup results are concatenated into a single dense format. The procedure typically requires an element-wise operation such as sum&pooling operation, and then are combined with dense continuous features. After that, the newly obtained dense data go through the rest of dense DNN training to improve model quality. The rest of the dense DNN models, including the fully-connected and activation layers, are referred to as the dense part.

The size of the sparse part is extremely large compared with the dense part. To achieve scalable distributed training, we adopt the model-parallel scheme for the sparse embedding tables and the data-parallel scheme for the dense parts. Figures 2.13 outlines the overall workflow of the default synchronous distributed training. In this work, we follow the design of Zion [66] to employ a decentralized approach for synchronous training, which mitigates potential network bottlenecks compared to the central PS design [67]. For the decentralized approach, parameter servers are colocated with the trainer processes. Specifically, the parameters for the dense part are replicated among the training processes to follow the data-parallel scheme. Correspondingly, the parameters for the sparse parts are split and distributed across multiple processes to implement a model-parallel scheme because the large size of the embedding table prevents model replication. For the dense part, an AllReduce operation is executed in the backward pass to synchronize the gradients obtained from the multi-node process on different mini-batches of the input data. Then, the partitioned embedding tables require an AlltoAll operation in the forward pass to distribute results after embedding lookup and in the backward pass to collect the

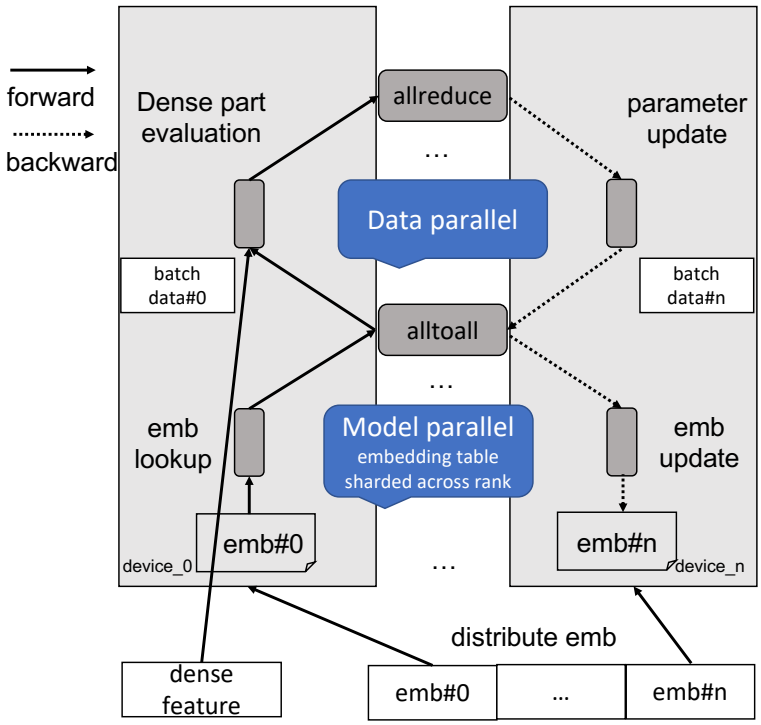
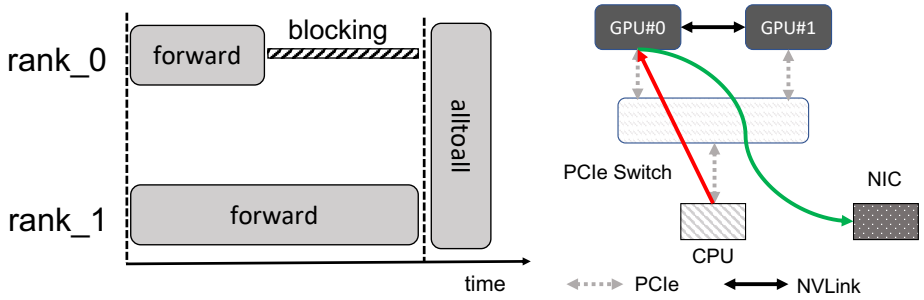



Figure 2.13: Distributed training of DLRMs.

gradients for the embedding table updates.

### 2.3.2 Challenges of Training Sparse Models

Typically, it is preferred to place the training computation jobs for deep neural networks in devices because GPUs can achieve up to petaflops of floating-point arithmetic (FP) operations for each GPU. At the same time, the sparse parts are placed on CPUs because of the large capacity of host memory. The fast development of modern accelerators equipped with larger device memory makes it possible to move partial embedding tables to GPUs. The training process can utilize the high memory bandwidth (HBM) of GPUs. Table 6.2 presents the features of the advanced NVIDIA A100 GPU, which has 40GB device memory. In addition, GPUs can deliver high throughput for larger batches, avoiding the additional CPU-GPU data transfer over PCIe. There are several challenges when deploying hierarchical embedding placement on modern hardware. PaddlePaddle supports direct device-to-device communications




(a) NVIDIA NCCL Alltoall. (b) Communication topology.

Figure 2.14: The green line represents the direct peer-to-peer GPU communications using GPUDirect-RDMA. The red line represents the data transfer between CPU and GPU.

without involving CPUs and host memory for multi-node training using GPUDirect-RDMA [68]. NVIDIA Collective Communication Library (NCCL) [45] implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. However, the Alltoall routines provided by NCCL cannot promise the best performance when training sparse models on the heterogeneous cluster. First, NCCL requires CUDA streams on each device to execute the communication operation, which occupies the precious compute capabilities. Second, the implementation of NCCL Alltoall is blocking and synchronous, which might cause network contention and increase transfer delay. As is shown in Figure 2.14a, though rank 0 has finished the forward pass for sparse parts earlier than rank 1, it has to wait for the completion of other ranks, then the Alltoall communication is initiated. This limitation motivates us to implement a non-blocking communication operator. In addition, the communication routines provided by NCCL only support on-device data. If the placement of the embedding table is hybrid, it requires additional data transfer between host and device, which can be illustrated as the red line in Figured 2.14b. These additional operations will introduce overhead on PCIe and occupy the memory bandwidth, increasing the communication delay. Last but not least, as for intra-node communication, Hybrid embedding placement also introduces non-trivial NUMA overheads

because maintaining cache coherence across shared memory has a significant overhead. Notice that NVSwitch provided by DGX-A100 enables full-bandwidth connectivity between GPUs. It is better to transfer data via high-speed NVLink instead of shared memory to reduce NUMA overhead further. A resource-efficient and asynchronous communication operator should be proposed.

Another challenge is high communication overhead during the training. Rapid increases in GPU compute capacity over time further shifts the bottleneck of training towards communication for recommendation models. The communication overhead for some models, computed as the percentage of total time spent on communication, is as high as nearly 60% due to expensive Alltoall and Allreduce communication [41]. Existing work such as Kraken [43] adopts asynchronous training to obtain higher training speed, which is hard to ensure good statistical efficiency. Pipeline parallelism is a general solution that keeps workers well utilized, combining pipelining with intra-batch parallelism. PipeSGD [69] proposes a pipelined training with a width of two that combines the best of both synchronous and asynchronous training. In this way, the expensive Allreduce can be overlapped with the computation. However, besides Allreduce, there are two more Alltoall operations and input data re-distribution, which requires a carefully designed scheme to partition the model and schedule the execution order. Furthermore, heterogeneous GPU-CPU clusters are particularly well suited to deep learning training because the CPU and GPU machines can work together to accelerate the training pipeline. We propose to move the I/O bound data re-distribution tasks to the remote CPU-only cluster after pipelining. In this way, the overhead on the sparse CPU resources and host memory bandwidth can be further alleviated. Although pipelining is not a new idea, enabling pipelining for recommendation model training requires fine-grained tasks scheduling and the insight of computation separation, which are unique features of Sven.

Last but not least, partitioning the embedding table across GPUs and CPUs is

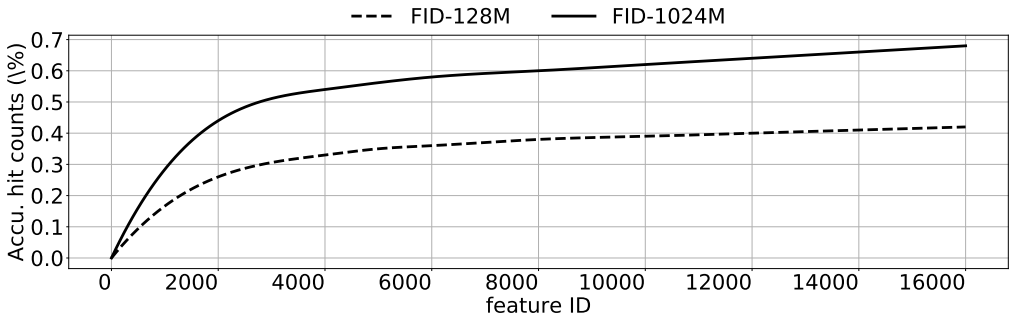



Figure 2.15: Access patterns of embedding tables.

another challenge when deploying a hierarchical placement scheme. The key design principle is that GPUs can take up more responsibilities for the embedding lookup and embedding update since they have higher memory bandwidth. In the meantime, we have to consider that the device's memory capacity is still very limited compared to hots memory. Therefore, it is important to understand the unique memory access patterns of embedding tables. We analyze the access distribution of embedding tables of Wide&Deep, which is summarized in Figure 2.15. We vary the cached embedding feature counts as 128M and 1024M, respectively. The x-axis of the figure represents the embedding feature accessed over the training recommendation models, while the y-axis shows the cumulative hit counts, which is normalized by the total number of embedding table request during the recommendation model training. It can be concluded from the results that access patterns to embedding tables follow the power-law distribution. When we increase the cached page size, the cumulative hit count is almost over 60%. It can be learned that the majority of embedding feature usage remains concentrated in a few hot memory regions. This observation motivates us to place those hot features on GPUs while the remaining CPUs, from which the forward and backward passes of sparse parts, can befit the high bandwidth of GPUs. Besides, an efficient entry replacement algorithm is required to ensure that the memory footprint is slower than the hardware constraint during the long-term training.

# CHAPTER 3: Related Work

## 3.1 GPU Memory Management Optimization for DNNs

A variety of solutions have been proposed to overcome the GPU DRAM shortage for training deep neural networks. Lossy encodings have been rigorously studied in the domain of DNN inference and training. Network pruning techniques [14] are proposed to reduce the model redundancy so as to reduce the memory consumption. Huffman encoding [15], quantization [70] and reduced precision [16] are also studied to reduce the model size (weights). Network compression [17] is another important approach to reduce the memory usage of DNNs. However, they provide limited opportunities for memory saving because weights are not a major contributor to the total memory requirement. Moreover, some of these techniques, e.g., reduced precision, might result in loss of prediction accuracy if not carefully tuned. Gist [18] investigates approaches to optimize the memory usage for input feature maps, which are the dominated source of memory footprint in DNNs training. This work is orthogonal to our approach.

SuperNeurons [20] and Chen et al. [71] introduce a recomputation strategy to trade computation for memory saving. They consider recomputing the output from selected layers in the forward again in the backward pass instead of prefetching them from CPU memory or keeping them on the GPU DRAM. However, it requires high-level semantics on the computation graph. The overhead from training cannot be negligible because the largest layers require longest recomputation time. This approach works well for linear networks but fail to exploit the memory saving opportunities. Yet, this technique can be applied in conjunction with our work for specific layers, e.g., batch normalization.

Model parallelism is a straightforward strategy to train deep and large networks.

DistBelief[22] distributes the network across multiple nodes by partitioning the network so that each node only holds a part of the the network. Tofu [55] enables the training of very large DNN models by partitioning a dataflow graph of tensors across multiple GPU devices. However, these techniques require huge intra-network communications for synchronization. Another approach is to place different layers on different devices via heuristics [72] or machine learning [73]. However,operator placement is not suitable for DNNs with a deep stack of layers. Data parallelism can achieve better performance by adding more GPUs. But the powerful GPU could suffer from sub-linear scaling because of stragglers and costly network transfer across workers. An effective and simple approach to reduce the memory requirement for DNN training is to reduce the minibatch size. However, it slows down the training process because a smaller batch size could result in GPU underutilization [74].

vDNN [19] also proposes a prefetching and offloading technique to transfer the data between CPU and GPU memory so as to fit the large networks in the GPU memory. It tries to overlap communication with computation by asynchronously swapping the data between CPU and GPU via PCIe. However, it requires a synchronization barrier between communication and computation for each layer, which is safe but inefficient. It ignores the benefit from those layers, e.g., POOL and ACT layers, which are cheap to compute. It is a waste to wait for the slow transfer of PCIe bus. SuperNeurons [20] also consider memory swapping but restricts to swap only convolution layers. None of these works take the GPU memory fragmentation into account when allocating the tensors on GPU. vDNN++ [75] proposes an approximated memory pool to reduce the GPU memory fragmentation, which is limited to linear neural networks only.

## 3.2 Asynchronous Distributed DNNs Training

A variety of solutions [21] have been proposed to overcome the straggler problem for distributed deep learning. Redundant execution [76][77] is commonly used to mitigate stragglers in the traditional data analytics platform. The main idea is to launch spec-

ulative execution on multiple machines. Recently, backup worker [29] is proposed in distributed learning systems to overcome the stragglers problem. However, the redundant execution introduces non-negligible overhead from data communication. Firstly, in ring All-Reduce, a more restrictive communication pattern makes it impossible to implement these techniques, e.g., backup works. Secondly, the redundant execution is not fruitful to handle the randomized system heterogeneity and inherent load imbalance.

Adaptive tuning strategy solves stragglers by matching the amounts of task loads to their respective capacities in a heterogeneous environment. FlexMap [78] launches elastic map tasks with dynamic input block sizes, and PIKACHU [79] is proposed to adjust the reduce task size elastically based on the system heterogeneity. However, all these works only focus on the traditional BSP scheme. Advanced approaches are proposed for deep learning systems. For example, $R^2SP$ [27] is proposed to tune the batch size adaptively and FlexPara [80] partitions parameters to provision adaptive tasks to match the varying capacity. However, these works do not fundamentally solve the problem because the severe and continuous slowdown of some workers will eventually drag down other workers and the whole training. In the operating system, work-stealing is a classical method to achieve load balancing among workers, improving system-wide performance. The concept of work-stealing is to move workloads from slower workers to the faster ones, e.g., FlexRR [81]. Skewtune [82] is proposed to mitigate skewness in the data analytics platform, which waits for idle workers to steal work from those tasks with the greatest remaining processing time. Considering the large overhead from communication, these approaches cannot be directly applied to deep learning systems. Recently, relaxed synchronization is proposed to exhibit the strict need for synchronization on the BSP model. SSP [25][30] enables processes to execute the training independently and allows fast workers to advance a bounded number of iterations ahead of slow workers. A-BSP [83] is proposed to aggressively

synchronize parameters by applying the partial updates from slower workers. But all these approaches target on the centralized PS architecture.

Taking advantage of the robustness of deep learning training process, AD-PSGD [34] is first to explore the fundamental algorithm level solution to allow asynchronous synchronization in a decentralized setting. In AD-PSGD, gradient updates are only sent to limited (random) neighborhoods using gossip algorithms. However, it requires extra overhead to perform atomic parameter averaging. Otherwise, it will suffer from deadlocks issues due to scheduling conflicts. Furthermore, the implementation is limited to a certain type of AllReduce graph. Similar approaches such as Cutout [84] and Dropout [85] propose random errors and omissions into the training process to improve generalization of network models. Hop [33] introduces a generic solution to overcome heterogeneity for decentralized training protocol, which proposes a queue-based synchronization mechanism to enable bounded staleness. However, maintaining a bunch of queues and tokens for each worker in a large cluster incurs communication overhead and delay. Furthermore, Hop accumulates gradients for faster workers. Due to the bounded iteration gap, some workers' severe and continuous slowdown will eventually drag down other workers and the entire training.

Prague [35] and Eager-SGD [86] are more related to our approach, which proposes a new communication primitive to allow partial workers to synchronize parameters quickly. Specifically, Prague offers both static and dynamic group scheduling to construct a new group randomly during the runtime to avoid conflicts. However, this approach is based on system profiling information, whose decision might not be optimal for dynamic neural networks such as RNN and LSTM. Moreover, it requires a careful group scheduling at each iteration to avoid the synchronization conflicts. It also introduces additional system overhead to form the communication graph, which harms the training throughput. Eager-SGD proposes solo and majority collective communication to implement an asynchronous decentralized SGD. Solo collective communication

could sacrifice model accuracy because it advances the synchronization aggressively. As is illustrated in the evaluation results, the majority cannot ensure the performance because it does not oversample enough to avoid the slower processes. Furthermore, in a heterogeneous environment, eager-SGD still suffers from a deterministic slowdown, which cannot be avoided by the randomized approach.

SGP [87] adopts a gossip algorithm called PushSum for approximate distributed averaging, which allows for much more loosely coupled communications to achieve efficient distributed training in a high-latency or high-variability environment. SGP does not use global collective communication primitives. Alternatively, each process only communicates with its neighbors. However, all the processes need to finish the current iteration before going to the next. SGP is robust to communication-constrained settings. Compared to SGP, our work is robust to load imbalance. RNA can relax the strict synchronization to tolerate computation straggler because of the feature of asynchrony. Both eager-SGD and RNA only require $\mathcal{O}(1)$ step to globally propagate the lo-cal update. However, in SGP, each process propagates its local update using $\mathcal{O}(logP)$ steps. Zero/DeepSpeed [88] presents a set of optimizations to reduce memory redundancy in distributed training, by partitioning parameter weights, activations, and optimizer state separately, and it can scale models to 170 billion parameters. Compared with Zero, RNA is straggler tolerant and is orthogonal to their approach.

## 3.3 System Innovations for Training Sparse Models

Various system-level innovations have been proposed and implemented to support larger and deeper neural network models. For example, DeepSpeed [88] proposed to shard model parameters, gradients, and optimizer parameters among all nodes and eliminate memory redundancies in data- and model-parallel training while retaining low communication volume to reduce memory usage. PipeDream [57] automatically partitions DNN training across workers, combining inter-batch pipelining with intra-batch parallelism to better overlap computation with communication. FlexFlow [54]

uses automatic search to discover the efficient parallelization strategies for DNN training. However, these systems do not specifically target sparse recommendation systems.

HugeCTR [89] is a high-efficiency GPU framework designed for Click-Through-Rate (CTR) estimating training provided by NVIDIA, which follows a synchronous training scheme. HugeCTR partitions the large embedding table across multiple devices to utilize the high memory bandwidth of the device to accelerate the training of recommendation models. However, HugeCTR is limited by the device memory size because they have to maintain all sparse features within devices. Sven combines GPU embedding with CPU embedding and proposes prioritizing hot features to utilize the high bandwidth of GPUs. Kraken's is another end-to-end training system [43] that implements a parameter server that dynamically adjusts the placement of sparse embedding tables. It also proposes an automatic feature admission scheme to ensure that the memory consumption is under reasonable constraints. However, its implementation is limited to the CPU. XDL is proposed to handle sparse features with very high dimensions. XDL [67] employs many techniques such as pipelining, sample compression, and NUMA binding to achieve higher training throughput of the deep learning recommendation models. Baidu proposes a hierarchical design [44] which proposes to use host memory to keep the out-of-device-memory parameter and employ SSDs to keep the out-of-host-memory neural network parameters for very wide and deep neural networks. Facebook DLRM [66] co-designs software and hardware to enable training models with trillions of parameters. On the software side, they implement a hierarchical memory architecture to host large embedding tables and leverage reduced precision training to reduce communication volumes. On the hardware side, they designed the ZionEX platform, which is equipped with a dedicated and high-speed network. Machines are configured with specific network transport and optimized network topology.

CHAPTER 4: Efficient GPU Memory Management for Nonlinear DNNs

## 4.1 System Design and Implementation

The design objective of our dynamic memory manager (Dymem) is to automatically manage the memory usage of DNNs while minimizing the overhead and maximizing the reduction of memory load. Dymem is a host-side runtime that interfaces with GPU to dynamically move, allocate, and release tensors. Figure 4.1 shows the overall system architecture. In this section, we first introduce how to perform graph analysis to construct a memory-efficient execution flow, particularly for nonlinear networks. Then, based on the results from the graph constructor, we propose a tensor scheduler to utilize dependency features to asynchronously offload/prefetch candidates with different variable sizes and resident duration. Lastly, we implement a unified GPU memory pool and propose a contiguity-conserving placement policy to allocate/deallocate the scheduled tensors.

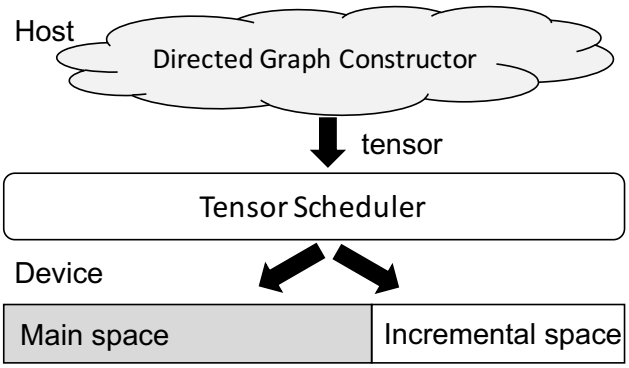
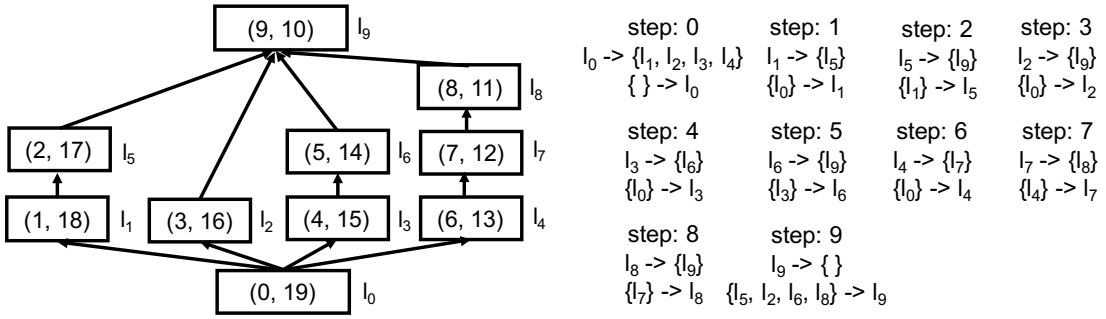



Figure 4.1: The system architecture of Dymem. Constructor performs graph analysis. Scheduler manages prefetch/offload operations. Allocator handles tensor allocation/deallocation.

**Algorithm 2** Execution flow for nonlinear blocks.

```
1: function flowConstruct(int layerId)
2:     if layerID == Null then
3:         return;
4:     end if
5:     refcnt++;
6:     if layerId.refcnt < prevLayer.refcnt then
7:         return;
8:     end if
9:     execFlow.push(layerId)
10:    L = layerId -> get-next();
11:    for l ∈ L do
12:        flowConstruct(l)
13:    end for
14: end function
```




(a) Schema of Inception-v4 network.

(b) Forward pass of Inception-v4.

Figure 4.2: Execution order for Inception-v4 network in the forward pass. $l_i$ represents $i_{th}$ layer. $l_0 \rightarrow \{l_1, l_2, l_3, l_4\}$ represents that layer $l_1$, $l_2$, $l_3$, $l_4$ have dependency on layer $l_0$.

### 4.1.1 Execution Graph Construction

Given a nonlinear network, we need a memory-efficient approach to set up the execution order. Since cuDNN [90] implements deep learning primitives at layer granularity, we use tensors as the basic scheduling unit. For basic networks, during the forward propagation, the results from $layer_{n-1}$ can be applied as the input for $layer_n$. The computation flow can be regarded as a sequential process. Only when the preceding layer is finished, then can it initiate the next layer's computation. This chain rule is similarly applied in the backward pass but in a reversed order. For

networks with nonlinear blocks, there are nonlinearities such as one-to-many (fork) and many-to-one (join) connections. Depth-First-Search (DFS) algorithm is used to decide the execution sequences for these nonlinear dependencies, which is shown in Algorithm 2. Whenever there is a fork connection, DFS is applied to explore all the executable layers until it reaches the join connection in the nonlinear blocks, as shown in lines 7 to 8. Figure 4.2a shows the schema for Inception-A blocks in the Inception-v4. The detailed execution order obtained by DFS is demonstrated in Figure 4.2b. In this example, the Inception block should be propagated in four branches in both the forward and backward passes. In the forward pass, $l_0 \rightarrow \{l_1, l_2, l_3, l_4\}$ represents that output feature maps from layer $l_0$ should reside in the GPU memory until layers $l_1$, $l_2$, $l_3$, $l_4$ are executed because of dependencies. Similarly, during the backward pass, in the branch $l_8 \rightarrow l_7 \rightarrow l_4$, when $l_8$ is being executed, layer $l_4$ should be prefetched from CPU memory asynchronously based on DFS. The reason why DFS should be applied to construct the execution graph lies in two properties: First, DFS requires less memory space to reach the join connection node in the nonlinear blocks when exploring the traversal path. For example, the branch $l_4 \rightarrow l_7 \rightarrow l_8$ illustrates simple dependency, in which the corresponding data for those memory-intensive convolution layers can be released from GPU memory sequentially. Second, inside those nonlinear blocks, e.g., residual block and Inception grid, most layers are computation-intensive Convolution layers. The DFS can serialize the sequences of convolution layers in each branch, mostly.

### 4.1.2 Dependency-aware Offloading and Prefetching

After obtaining the execution graph, Dymem automatically manages the offload and release operations for tensors so as to effectively improve the overlap ratio between communication and computation. We employ two separate *cudaStreams* to transfer tensors in/out of external memory asynchronously. $stream_{compute}$ interfaces to the cuDNN handle and sequences all the computations in the forward and backward

Table 4.1: The computation and communication time in residual block (ms).

| Layer | 1×1 CONV | 3×3 CONV | Join |
|---|---|---|---|
| Computation | 25 | 76 | 3 |
| Communication | 66 | 68 | × |

pass. $stream_{memory}$ is responsible for the tensor placement, movement, allocation, and deallocation.

#### 4.1.2.1 Memory Offload

During forward propagation, if $layer_n$ is available for offloading, Dymem first allocates a pinned memory region in the host via $cudaMallocHost()$, then $stream_{memory}$ can asynchronously swap feature maps from this layer via non-blocking memory transfer. When the asynchronous offload is completed, the $cudaEvent$ is register to record this event. Because the input features for CONV, POOL and ACTV layers are read-only data structures, we can start the offload operation for these when they are being performed forward propagation. As for $stream_{compute}$, $layer_n$'s computation can be started as soon as $layer_{n-1}$'s computation is completed without waiting the completion of the offload operation of $layer_{n-1}$. Nonlinear blocks, e.g., Residual blocks, can benefit from this strategy because of the join operations do not necessarily wait for the completion of tensors transfer from 1×1 CONV and 3×3 CONV, which is illustrated from the Table 4.1. $stream_{compute}$ guarantees the completion of computation for $layer_n$ by using the $cudaStreamSynchronize()$ API. When both of these two events for $layer_n$ are finished, a shared queue is used to record this tensor. The release of the tensors chosen for offloading from GPU is done when there is no dependency for these layers in the shared queue. An individual thread is launched to release the $layer_n$ from the GPU memory. At the end of the forward propagation, we synchronize $stream_{compute}$ and $stream_{memory}$ to make sure that $stream_{memory}$ has offloaded its feature maps. This safely ensures that all layers chosen to be offloaded

are offloaded from GPU memory before the start of backward propagation, maximizing the memory saving and improving the performance greedily. However, there is an exception that the execution for the next layer has to be blocked if the available memory is not enough, waiting for the release for completed layers. In general, memory space is traded for performance in the forward pass.

#### 4.1.2.2 Memory Prefetch

In the backward pass, prefetching the offloaded input feature maps back to GPU can be overlapped with the computation of backward pass using $cudaMemcpyAsync()$ as well. After an asynchronous transfer for $layer_n$ is completed, a $cudaEvent$ is registered in the $stream_{memory}$, after which the computation can be started for this layer. The $stream_{compute}$ is synchronized with the offload event to guarantee that the computation can be safely launched with available input feature maps. Similar to the forward pass, we only synchronize $stream_{compute}$ and $stream_{memory}$ at the end of backward propagation before the next iteration. Instead of launching the prefetch operations in the reverse order simply, we have to consider the execution order and prefetch latency when searching for the optimal candidate layer. Another problem is that if the prefetched $layer_m$ is too far away from the overlapped $layer_n$ , the memory saving benefit will be reduced because the prefetched data be reused immediately, wasting the GPU memory. Jointly considering the memory saving and prefetch latency, we propose an efficient searching algorithm to decide the layer to be prefetched, which is shown in Algorithm 3. Whenever there is a nonlinear block, we decide the preceding layers based on DFS, which is similar to the forward pass. After obtaining the layer, we restrict that no more than two Convolution layers residing in the GPU, as is illustrated in the line 11. This is because Convolution layers are computation intensive. Prefetching these layers too early will under-utilize the GPU resources. As long as it is not convolution layer and not available yet in the GPU memory, it can be chosen as the candidate, as is shown in line 14. This is because

other layers require shorter computation time compared with Convolution layers. This feature can gain performance improvement because the prefetch latency can be well hidden by the computation time.

**Algorithm 3** Searching the candidate layer.

```
1: function searchPrefetchLayer(int layerId)
2:     n = 0;
3:     if layerId-> type == CONV then
4:         n++;
5:     end if
6:     next = flowConstruct(layerId).pop();
7:     while id do
8:         if next-> type == CONV && n < 2 then
9:             pf.push(id); n++;
10:            next-> pf = True;
11:        else if next-> of && !(next-> pf) && next-> type != CONV then
12:            next-> pf = True;
13:            pf.push(next);
14:        end if
15:        next = flowConstruct(next).pop();
16:    end while
17: end function
```

### 4.1.3 Contiguity-conserving Memory Management

In this section, we first define tensor mobility based on the varied data size and dynamic dependencies. Then we propose Group Tensors By Mobility (GTBM) as the placement policy to classify tensors before allocation. We further implement a unified memory pool, consisting of main space and incremental space, to host different tensors so as to achieve lower memory fragmentation.

#### 4.1.3.1 Design Principles

In the operating system, the internal fragmentation is defined as the inability to satisfy an allocation request because a suitably large contiguous block of memory is not free even though enough memory may be free overall [91]. The scope of internal fragmentation not only depends on the layout of free memory nodes but the size of

the request. Here we define the unusable free space term, $U_f$. It measures how much of the available free memory cannot be used to satisfy an allocation:

$$U_f(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}, \tag{4.1}$$

in which $j$ is the desired allocation (i.e., the size of the request is $2^j$), $TotalFree$ is the number of free memory nodes, $2^n$ is the largest request allocation that can be satisfied, and $k_i$ is the number of free memory nodes of size $2^i$. A term of 0 implies there is no memory fragmentation. The term tending towards 1 implies high fragmentation, in which the request cannot be satisfied. Based on the analysis of the best-fit algorithm in Section 2.1.3.2, the memory fragmentation could increase if the contiguous and noncontiguous tensors are both allocated to a contiguous portion of the virtual address. If fragmentation happens, the memory pool has to grow its size so as to satisfy the demand. In order to decrease $U_f$ during the training, we propose a contiguity-conserving allocation strategy. The core idea is to provide a soft guarantee that all of the tensors having the same dependencies or similar lifetime should be allocated in the same region.

#### 4.1.3.2 Group Tensors By Mobility

Group Tensors By Mobility (GTBM) considers the address space as being split into three arenas. Tensors are placed such that each arena contains tensors of the same page mobility type. For our purposes, three mobility types are defined as below:

- **Movable tensors** are from those layers who have simple dependencies, i.e., one reference count. They can be released after the being propagated and the released space could coalesce with each other soon.

- **Temporary tensors** are those tensors that are known to exist for a very short period of time, such as fork connections or tensors waiting to be joined. These

tensors exist longer than movable tensors and are not supposed to be mixed with them.

- **Reclaimed tensors** are tensors from layers that are not selected for offloading. These tensors require less memory capacity and could be reclaimed after being propagated.

The placement policy is to group tensors of the same mobility type within an arena of the matching type. The pseudo-code for the grouping process is summarized in Algorithm 4. The grouping procedure for Dropout, Softmax, and FC layers are shown in the line 5, which should be classified as reclaimed tensors. For the nonlinear block shown in Figure 4.2, in the forward pass, layer $l_0$ has multiple dependencies, which should be put into the temporary tensors group. But for layers $l_5$, $l_2$, and $l_6$, though they have only one reference, they should be grouped as the temporary tensors because their results will not immediately be concatenated, which is illustrated in the line 5. However, in the backward pass, they should be regarded as movable tensors because the results from layers $l_5$, $l_2$, and $l_6$ can be consumed immediately. The release operations will be initiated when the reference number of the currently processing layers has been decreased to zero. For example, after forward computation of $l_9$ is finished, the feature maps of layers $l_5$, $l_2$, and $l_6$ can be released from the GPU.

#### 4.1.3.3 Unified Memory Pool

The memory manager is a host-side interface, serving as the GPU back-end. The memory space is divided into main and incremental areas. Inside the main space, the allocation can be started from both the low and high-end available addresses. For the memory operations, we employ the open-source asynchronous memory allocation/release API library provided by Nvidia CNMeM [51]. The allocation starting from the low end will use the default $cnmemMalloc()$ API, following the default best-fit heuristic. For the allocation starting from high-end address, we implement a new

*cnmemHighMalloc*() API, pointing the starting address to the high end. For the incremental space, we use the default *cudaMalloc*() to request a new GPU memory outside of the existing pool. Though this procedure will cause initialization overhead, it is negligible because of the minimal proportion of these layers in the neural networks.

For movable tensors, Dymem allocates tensor for the subgraph in the memory pool via *cnmemMalloc*() from the low-end available address. Since they have a simple dependency, i.e., one dependent layer, it only resides on the GPU for one step and then will be offloaded to the CPU memory. In the forward/backward pass, the released space in the low-end address can always be utilized by the coming layers if the communication is well hided or the subsequent layers. As for temporary tensors, Dymem places the coming tensors via *cnmemHighMalloc*() starting from the high-end available address. In this scenario, contiguity is conserved in the high-end address since all of the tensors from this group have noncontiguous usage. Inside the nonlinear blocks, the tensors that have higher reference counts are always allocated before the ones with lower references. So the used space can be deallocated in an order opposite to the allocation, resulting in minimal memory fragmentation. Regarding the incremental space, which hosts reclaimed tensors, because the backward propagation follows the reversed order of forward pass, the allocation is from low to high-end while the deallocation is from the high to low-end, leading to no fragmentation.

The configuration of memory pool size is a trial-and-error process. To ensure the trainability of networks, Dymem initiates a large enough main space for the first iteration. Based on the aggregated consumption of memory, Dymem adjusts the memory pool size by removing the smallest squeeze gap between the low- and high-end regions in training, i.e., $mem\_size = (init - squeeze\_gap) + \beta$, in which $squeeze\_gap = \min\{highest\_avail - lowest\_avail\}$. The $\beta$ is reserved space in case of fluctuation. If the training fails, an additional $\beta$ is provisioned.

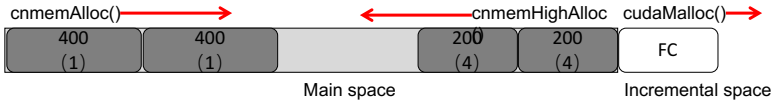
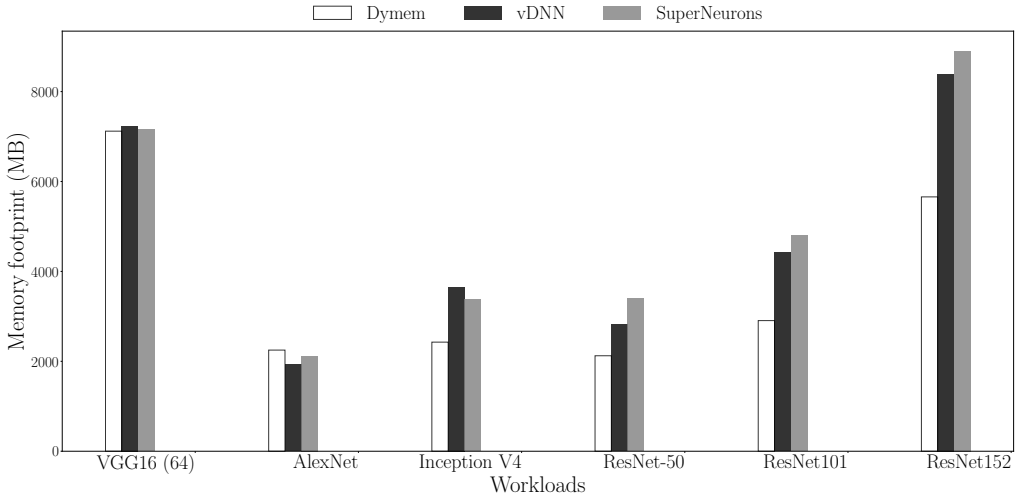
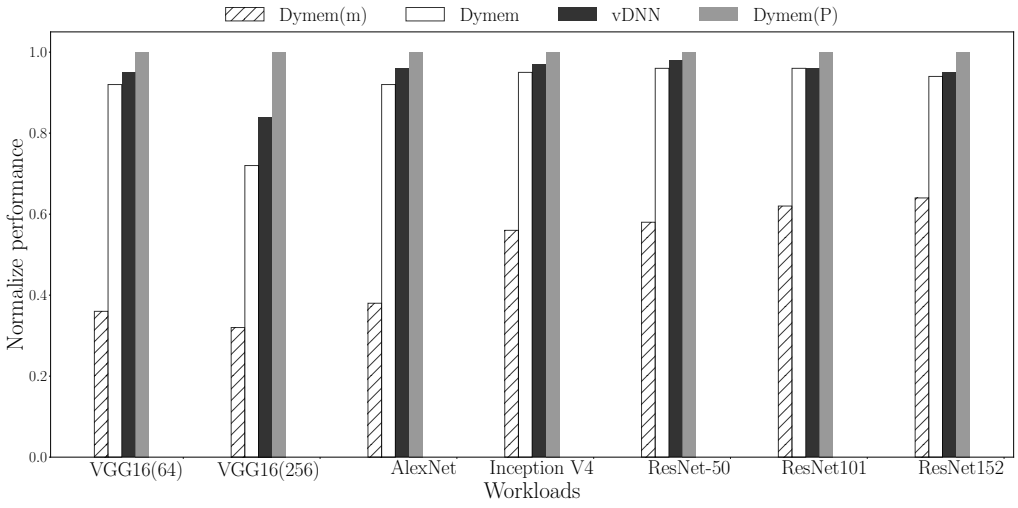



Figure 4.3: The workflow of the unified memory pool. Dark blocks represent main space and white blocks represent incremental space.




(a) Maximum memory usage.




(b) Normalized convolutional performance.

Figure 4.4: Overall GPU memory usage and normalized performance of convolution layers. The batch size of Inception V4 is 128. The batch size for ResNet with different depths are 100. Dymem(m) and Dymem(p) represent running Dymem with memory-optimal and performance-optimal algorithms.

**Algorithm 4** Group Tensors By Mobility.

```
1: function tensorGroup(int layerId)
2:     if layerId -> type == FC || layerId -> type == Softmax || layerId -> type
   == Dropout then
3:         reclaim ← (layerId);
4:     else if layerId -> refcnt > 1 || !(layerId -> get-next())  then
5:         temporary ← (layerId);
6:     else
7:         movable ← (layerId);
8:     end if
9: end function
```

## 4.2 Methodology

### 4.2.1 Baselines

We choose vDNN [19] and SuperNeurons [20] as the baselines for performance comparisons. Regarding memory management, vDNN uses the default Nvidia CNMeM [51] library to allocate/deallocate tensors. SuperNeurons adopts a fast heap-based GPU memory pool utility. The core concept is to divide the preallocated pool into an allocated list and an empty list. For these two techniques, we implement the best-fit algorithm as the memory management policy. The execution order for the nonlinear network is not detailed in vDNN. So we adopt the same construction, DFS, for vDNN for comparison. We can only release tensors from GPU memory when there is no further reference in the forward or backward passes. As for the tensor scheduling policy, we implement the dynamic policy mentioned in the vDNN paper, which automatically decides the offloading layers employed to balance the trainability and performance of a DNN at runtime. As for SuperNeurons, we only implement the liveness analysis and unified tensor management components because recomputation for specific layers is not considered in our work.

### 4.2.2 Optimizing Convolution Algorithm

The speed of CONV layers significantly impacts the training performance. We implement a dynamic strategy to utilize the availability of the GPU memory pool. The dynamism can achieve a tradeoff between memory saving and performance gain. Since the allocation of convolution workspace does not affect the functionality of the training, so we prioritize the allocation for those required feature maps, gradients maps, and weights, etc. The runtime will profile the available memory space when it enters a new layer and increment the amount of GPU memory for workspace in a fine-grained granularity, i.e., 1MB. The runtime will stop requesting more GPU memory for workspace if it causes failure in training. Then, the corresponding decision is regarded as the optimal configuration for the current state. The baselines, vDNN and SuperNeurons, adopt the same dynamic strategy to achieve the balance between memory saving and performance gain. We also implement the memory-optimal algorithm as the baseline, in which no extra workspace memory is required for Convolution layers.

### 4.2.3 DNN Benchmarks

#### 4.2.3.1 Linear Networks

First, we perform the evaluation compared with vDNN and SuperNeurons on linear networks, VGG-16 and AlexNet. We use the same training configurations as the published paper [10][92]. For AlexNet, we configure the batch size as 256. there are 23 forward steps and 23 backward steps. VGG-16 is one of the largest and deepest DNN architecture, which has 16 CONV and 3 FC layers. It requires substantial memory capacity for trainability. To ensure the trainability, we configure the batch size as 64 and 128. We evaluate the performance regression of the end-to-end training and the peak memory consumption for one iteration. Since SuperNeurons requests a fixed and large enough memory pool, we measure memory usage in terms of aggregated

memory usage during the runtime.

#### 4.2.3.2 Nonlinear Networks

We further perform the evaluation against vDNN and SuperNeurons on two representative nonlinear networks, ResNet [49] and Inception V4 [8]. Specifically, we implement the basic Residual block, which has two 3×3 convolutional layers with the same number of output channels. Each convolution layer is followed by a batch normalization layer and a ReLU activation function. Then, the skip connection joins the output from two convolution layers with the original input before the final activation function. We also evaluate the performance using various depths for ResNet, e.g. ResNet-32, ResNet-50, ResNet-101 and ResNet-152. The difference among these networks is the number of residual blocks. Since vDNN does not report the evaluation results for ResNet, we follow the implementation from Torch [59] to implement the memory management policy because it is adopted as the baseline for vDNN. For all of the above benchmarks, we use the image dataset CIFAR-10 [93].

## 4.3 Evaluation

Our experimental evaluation is performed on GeForce GTX TITAN X with 12 GB GPU memory. The machine has 3.4 GHz Intel i7-3770 CPU (20 cores) and 32 GB CPU memory. The GPU communicates with CPU via a PCIe switch, which has 16GB/sec data transfer bandwidth. The machine is installed with Ubuntu-16.04, CUDA 9.0, CuDNN 7.0, and g++ 5.4.0.

### 4.3.1 Reduction on GPU memory usage

Figure 4.4a summaries the aggregated memory usage among Dymem, vDNN, and SuperNeurons for different DNNs. Because all of these three approaches apply a layer-wise memory allocation policy, the GPU memory usage during forward/backward pass will fluctuate depending on the tensors chosen for offloading/prefetching. So we use the aggregated memory to represent the maximum allocated GPU memory

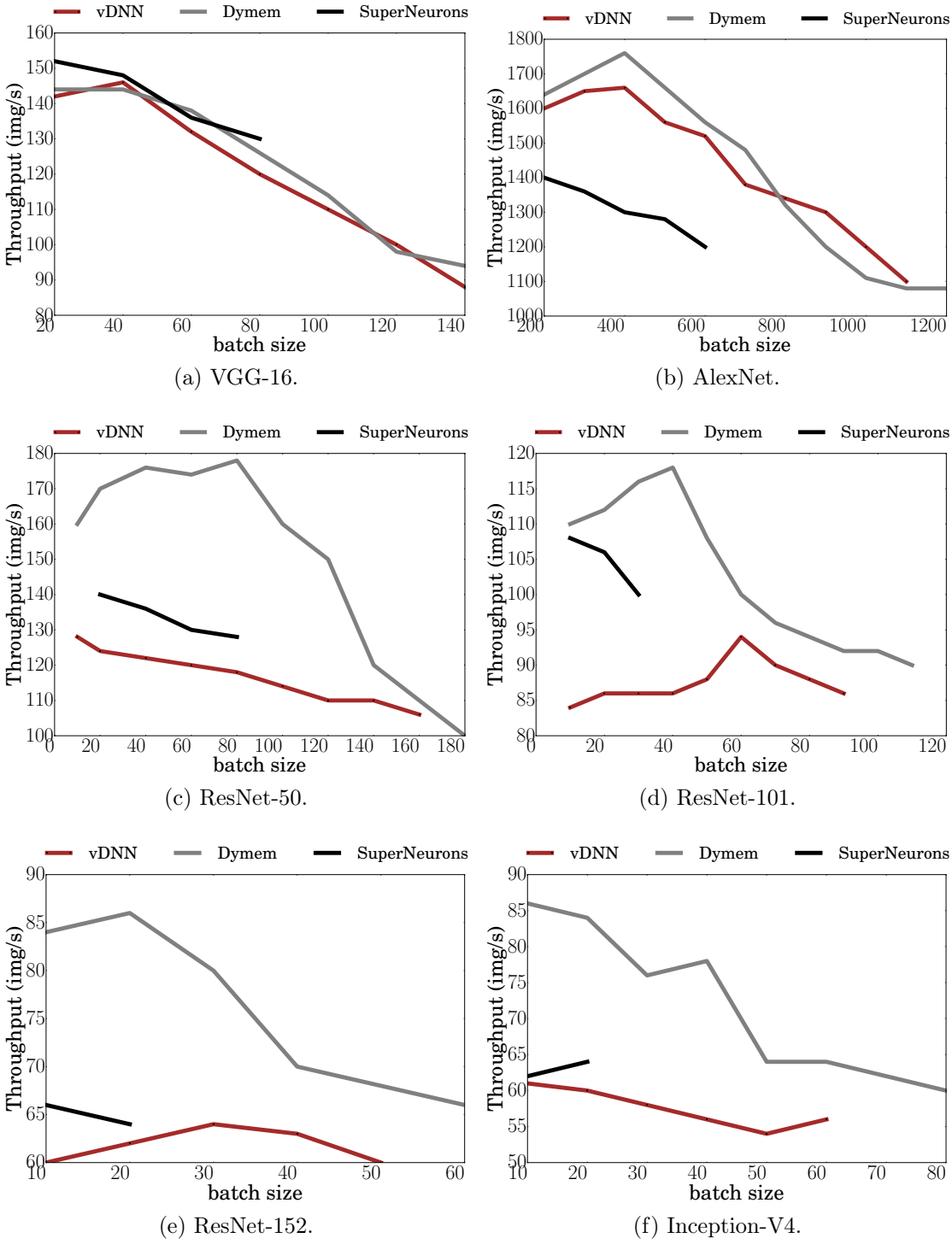
for one entire iteration, which is the minimum requirement to enable the trainability of the networks. From the results of VGG-16 and AlexNet, we can see that there is no difference between these three strategies. For such linear networks, layers are propagated sequentially. Dymem falls back to the same best-fit algorithm, which is adopted by SuperNeurons and vDNN as well. For VGG-16, because convolution layers dominate the training process, leading to no difference between vDNN and SuperNeurons. Dymem even requires nearly 250MB more memory capacity than vDNN for AlexNet. Because Dymem aggressively prefetches more layer's data structure than vDNN, trading the memory space for performance improvement, which is detailed in Section 4.3.2. For nonlinear networks, ResNet-50, ResNet-101 and ResNet-152 vary their network depths by changing the combinations of for-loop residual blocks. We can see that when the depth of ResNet is increased, memory consumption is not linearly increased. The performance of Dymem shows considerable scalability for different depths of networks compared with vDNN and SuperNeurons. Specifically, for Inception V4, the maximum memory footprint is reduced from 3650MB to 2527MB, resulting in 31% memory saving compared with vDNN. SuperNeurons shows better performance than vDNN to handle the tensor scheduling for Inception V4, whose inception branches are much more complex than ResNet. However, Dymem can still achieve 28% memory saving compared with SuperNeurons by minimizing the GPU memory fragmentation caused by the simple best-fit policy.

Figure 4.4b reports the performance of the convolution algorithms of Dymem and vDNN. The performance of convolution algorithms can better represent the average utilization of the GPU DRAM during the runtime. For comparison, we implement Dymen with both memory-optimal algorithm and the performance-optimal convolution algorithm as the baseline, in which performance-optimal algorithm is configured to supply with enough memory to run the fastest convolution algorithms. To demonstrate the impact of stressed memory capacity, we especially study VGG-16 running

with 256 batch sizes. Since is impossible to train VGG-16 with this configuration on GeForce GTX TITAN X, we employ the layer-by-layer strategy to ensure trainability. The training time that occurred in all convolution layers is accumulated to represent the overall performance because the memory capacity only affects the convolutional performance. As shown in the figure, we can see that the memory-optimal algorithm could result in nearly 60% performance loss on average compared with the performance-optimal algorithms. It is normal because no extra memory space is sacrificed for performance, closing the gap between the memory-optimal and performance-optimal configurations. Both Dymem and vDNN achieve well balancing between memory usage and the overall performance. From the results, we can see that Dymem and vDNN reach an average of 95% and 97% throughput of the performance-optimal. However, when the batch size of VGG-16 is configured to 256, the average throughput is decreased to 84% running in the vDNN. The performance is worse running in Dymem, which is 72% of the performance-optimal setting. Because Dymem prioritizes functionality over performance. For VGG-16, especially in the backward pass, Dymem prefetch more Convolution layers following Pooling layers than vDNN, resulting in less available GPU DRAM for workspace allocation. The minor performance loss from the algorithm could be made up of the benefit from the overlapping between computation and communication, as illustrated in Section 4.3.2.

### 4.3.2 End-to-end throughput evaluation

Figures 4.5 present the end-to-end training throughput comparison of Dymem to vDNN and SuperNeurons. The training throughput is measured by the number of processed images per second. We vary the batch sizes for different DNNs and compare the corresponding throughput. For linear networks, VGG-16, and AlexNet, there is not much performance improvement over vDNN and SuperNeurons. Because these networks are composed of simple and sequential layers. For example, VGG-16 consists of 16 convolution layers, which are computation-intensive. The computation




(a) VGG-16.

(b) AlexNet.

(c) ResNet-50.

(d) ResNet-101.

(e) ResNet-152.

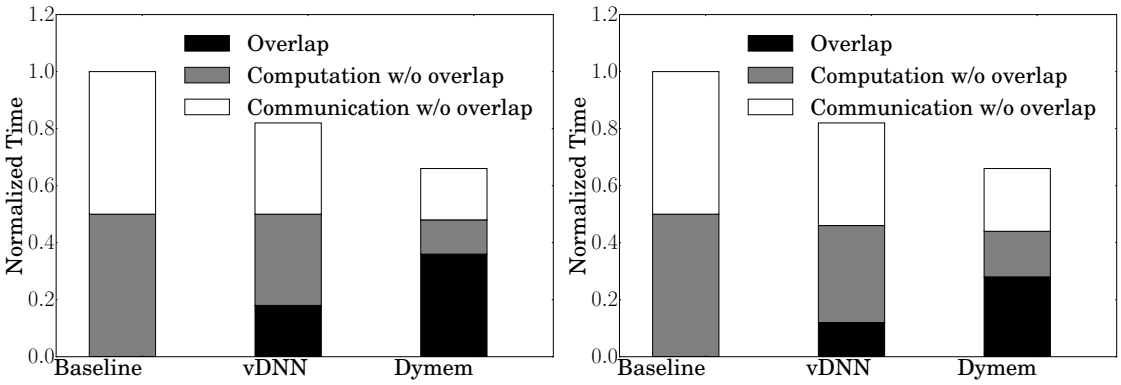
(f) Inception-V4.

Figure 4.5: End-to-end evaluation on throughput of different DNN models.

time is always longer than the transfer. The propagation computation dominates the total delay. As a result, there is no much performance benefit by removing the layer-by-layer synchronization barriers. In some cases, for linear networks, we can see that SuperNeurons perform better than Dymem and vDNN. Because SuperNeurons only offloads convolution layers, avoiding the communication overhead. However, for both linear and nonlinear networks, when the batch sizes are increased, SuperNeurons cannot train these networks because of the limited memory availability. For nonlinear networks, the results consistently demonstrate the leading throughput on ResNet-50, ResNet-101, ResNet-152, and Inception V4. The largest throughput improvement comes from ResNet-50, running with batch size 100, which achieves up to 42% compared with vDNN. The performance largely results from the improved communication/computation ratio. This is because Dymem could better utilize the overlap of communication and computation among layers. We can also observe that the throughput has slowly deteriorated by increasing batch size. This is because GPU memory can only accommodate less network layer with wider networks, resulting in the decreased communication/computation ratio. Less layer overlapping requires the growing communications in more frequent tensor swapping between CPU and GPU. Then, the runtime has to constantly offload the current layer before proceeding to the next one.

### 4.3.3 Efficiency of dependency-aware swapping

Figure 4.6 plots the breakdown of the normalized execution time of two representative nonlinear networks, Inception V4 and Resnet-32. These two networks are training on Dymem and vDNN with the memory-optimal configuration to avoid the impact from the speedup of Convolution. Specifically, the time is decomposed into the overlapped time, the non-overlapped communication time, and the non-overlapped computation time. In this experiment, the baseline only uses one stream, which restricts that the computation and offload/prefetch in both the forward and backward




(a) Inception V4. (b) ResNet-32.

Figure 4.6: Execution time decomposed into the overlapped time, the non-overlapped communication time, and the non-overlapped computation time in two networks.

passes are executed sequentially. We also configure the memory-optimal algorithm for these three experiments, so as to avoid the impact of the dynamics in the convolution layers. As shown in the figures, the overlapped time in the baseline is zero since the communication and the computation are performed sequentially. The layer by layer strategy adopted by vDNN can overlap the communication with the computation to some extent by 18% and 12% for Inception V4 and ResNet, respectively. The overlapped time in Dymem is longer than that in the vDNN, showing that a more aggressive batching strategy is more effective in terms of performance. As a result, compared with the baseline, Dymem can achieve nearly up to 46% reduction on the execution time.

### 4.3.4 Efficiency of the contiguity-conserving policy

To study the efficiency of the proposed tensors placement policy, we analyze the step-by-step memory usage of ResNet-32 in one iteration running with GTBM and the default best-fit algorithms, which is shown in Figure 4.7. Here, the "used" memory includes the allocated and the free but unusable memory nodes. For vDNN, the "used" memory counts the memory nodes from the lowest allocated address to the highest available address in the memory pool. For Dymem, it additionally counts

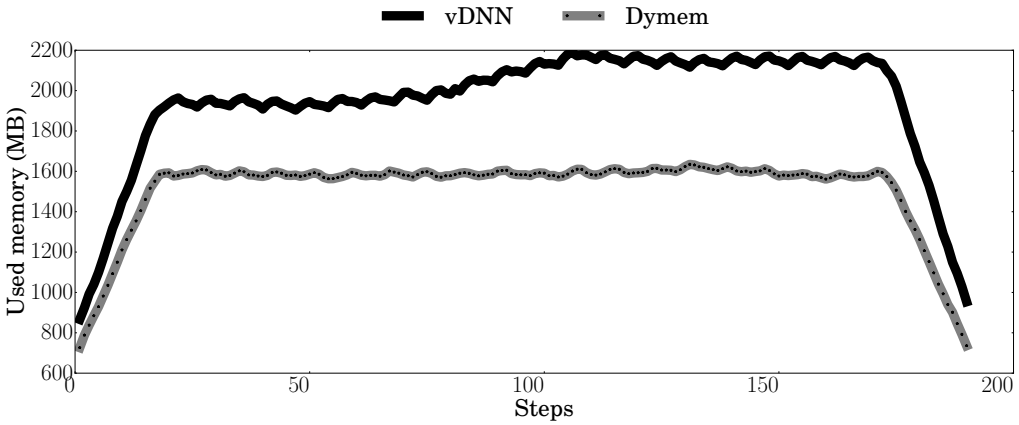



Figure 4.7: The moving average memory usage of ResNet-32 in one iteration.

the used memory nodes from the high-end in the main space and the incremental space. In this experiment, we run ResNet-32 without batch normalization under the batch size of 100. Specifically, it consists of 15 residual blocks. In each residual block, there is one join and one fork connections. Between these two connections, there are two branches, including shortcut and residual connections. The residual connection is composed of two Convolution and one Activation layers. After the join connection, the result will be fed into one Activation layer. One iteration requires 190 steps in the forward and backward passes. From the result, we observe vDNN requests more GPU memory at the end of each residual block, i.e., the join connection. Though the occupied memory nodes have been released, the fragmented but unfit memory nodes cause a waste of resources. Compared with vDNN, the unifies memory pool can reduce the fragmented nodes by reorganizing the placement. The peak memory usage occurs in the first residual block in the backward pass, which requires nearly 2500MB GPU memory for vDNN. But for Dymem, the peak memory requirement is 1854MB. The proposed unified memory pool can reduce memory fragmentation.

### 4.3.5 Sensitivity analysis of the approximation

To quantify the effect of different $\beta$, we configure $\beta$ varying from 100 to 500MB and run ResNet-101. We initiate a large enough memory pool, i.e., 10G (another

Table 4.2: The failure rate of training with different parameter configurations.

| Memory size (MB) | 100 | 300 | 500 |
|---|---|---|---|
| Training failure (%) | 5 | 0.9 | 0 |

2GB for incremental space), for the first iteration. Then we approximate the suitable memory pool size based on the profiled data and $\beta$. If the training fails, the current iteration is restarted and assigned with additional $\beta$ memory space. We obtain the average failure rate, which is shown in Table 4.2. Since the DNN training remains the same execution sequences, mostly, the approximated pool size is sufficient to serve the memory request. In this experiment, we can see that the optimal configuration should be 300MB, considering the memory-saving and network trainability. The limitation of this profiling-based method is that it requires additional memory and CPU capacities. For different networks with different parameters, the approximation should be repeated to find out the suitable configuration.

## 4.4 Summary

With the deep neural networks going wider and deeper, there is a need to effectively schedule GPU memory for DNN training to overcome the insufficient capacity. This work focus on memory management for the training of nonlinear DNNs. I propose the runtime to adopt the layer-wise graph analysis and dependency-aware memory offloading/prefetching strategy to improve the throughput of DNN training. Furthermore, I design a Group Tensors By Mobility (GTBM) placement policy to allocate tensors on the proposed unified memory pool for data structures with varied data sizes and dynamic dependencies, so as to reduce the GPU memory fragmentation in the training. Compared with the state-of-art vDNN, for linear networks, there is no much performance difference. For nonlinear networks, our proposed solution can achieve memory saving for Inception V4 by up to 31%. The proposed dependency-aware approach can improve the end-to-end training throughput for ResNet-50 by up

to 42%. The experiments also show that Dymem can achieve better scalability for nonlinear networks with various network depths. Currently, the proposed solution only supports GPU memory optimization for neural networks with a static dataflow graph and a fixed shape of the input, i.e., DNN.

# CHAPTER 5: Mitigating Stragglers in the Decentralized Training

## 5.1 Design of Randomized Non-blocking AllReduce

Traditional synchronous operations in the decentralized training such as AllReduce require a central scheduler to maintain a complete view of all processes. The synchronization operations can only be initiated after the slowest process finishes its computation. In this section, we introduce Randomized Non-blocking AllReduce (RNA), which takes a radically different approach: all processes operate in parallel, and the central scheduler does not maintain any progress state about training. The central scheduler relies on instantaneous progress information acquired from worker machines to initiate a synchronization. When the synchronization is triggered, RNA adopts weighted averaging to local accumulated results and dynamic scaling to the global aggregation to apply the obtained gradients.

### 5.1.1 Randomized Partial Collectives

The key to avoiding the "long-tail effects" for Ring AllReduce enforce partial collective operations, which force the slow processes to execute the synchronization. The main difference between the bulk synchronous parallel collective communication and the partial collective communication is when synchronization is triggered. For bulk synchronous parallel, the AllReduce is executed when all workers inform the central scheduler that they are ready to reduce the obtained gradients, e.g., $NEGOTIATE\_ALLREDUCE$ in Horovod. In this scenario, even a single delayed process affects the job's training time. In contrast to the synchronous mode, in MPI, there is a wait-free operation, which is called partial collective communication [94]. It forces the slow processes to execute the collective communication as soon as there

is one process executing it. This process, called the initiator, is in charge of enforcing the others to join the collective communication. In partial collective communication, an *external* activation is allowed to enforce a process to execute the synchronization before it reaches the *internal* activation. The time when the synchronization should be initiated is a trade-off between system and algorithm efficiency. A simple and straightforward strategy is to select an initiator among processes randomly. When a process is elected as the initiator, if it has gradients ready to be reduced, an external activation is broadcast to all the other processes to join the collective operation, regardless of they have finished the propagation or not.

Ideally, random selection can guarantee that at least half of the processes on average can take part in the collective operation and contribute their gradients. For a cluster with $N$ nodes, the probability that any process is selected as the synchronization initiator is $\frac{1}{N}$. Correspondingly, half of the processes on average have gradients ready for AllReduce before the selected initiator sends out the external activation. However, for a workload with a long tail distribution, e.g., as is illustrated in previous section, stragglers still have a high probability of slowing down the synchronization. Specifically, the expected waiting time is $\frac{1}{1-\rho}$, when there is workload in the queueing system [95], in which $\rho$ represents the computational load.

### 5.1.2 Per-process Sampling

Inspired by the power of two choices load balancing techniques [37], RNA implements a power of two choices technique to improve the purely random selection of initiator of AllReduce. It provides low expected waiting time using a stateless, randomized approach. More precisely, For a fixed time $T$, using $q$ choices, i.e., two here, the waiting time in an initially empty system over $T$ is upper bounded by $\sum_{i=1}^{\infty} \rho^{\frac{q^i-q}{q-1}} - \mathcal{O}(1)$, which improves the expected waiting time exponentially compared to the random strategy. The term $\mathcal{O}(1)$ is obtained in an initially empty system over the first $T$ units, which may depend on $T$. The central scheduler randomly selects

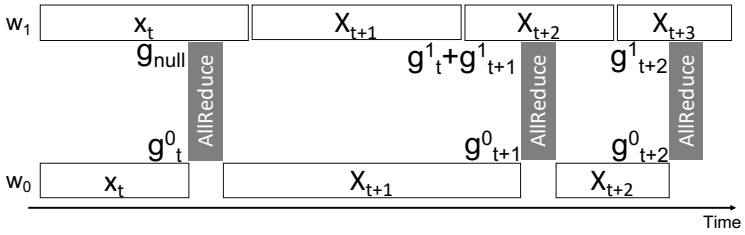



Figure 5.1: Non-blocking AllReduce: white bars represent computation processes, while grey bars represent communication processes. $x_t$ represents the parameters being used for training, and $g_t^0$ represents the gradient from processes $w_0$ at time $t$.

two processes among the machines and sends a *probe* to each, where a probe is a lightweight RPC. The selected processes can only reply to the probe when they have ready gradients. As long as one of them responds to the central scheduler, the AllReduce operation is initiated. The probe is attached to the iteration identification to avoid the scheduling conflict. When the faster one is replied, another probe is expired. For example, if processes $p_i$ and $p_j$ are selected and finish propagation for the current iteration at the same time, they both reply to the probe. There are two cases: 1) response from $p_i$ has been accepted, the probe for this iteration for $p_j$ is expired; 2) no response has been accepted, faster $p_i$ is accepted and, the probe identification is updated to the next iteration. Two-probe sampling can reduce the response time effectively compared with randomized approach. An additional number of probes cannot improve the performance but harm the performance because of the system overhead from sampling and messaging, which is detailed in Section 5.6.4.

### 5.1.3 Non-blocking AllReduce

**Algorithm 5** Non-blocking AllReduce.

**Require:** A set of workers $M$; the communication topology $G$.

1: **for** worker $m_i \in M$ **do**
2: compute the gradient $g_{k,i} = \nabla_{\varepsilon_{k,i}} f(x_{k,i}; \xi_{k,i})$
3: obtain the weight for gradients $W = \frac{1}{\sum w_{k,i}}$
4: average gradients using Non-blocking AllReduce $\overline{g}_k \leftarrow W \cdot \sum_{i \in M} g_{k,i}$
5: update parameters $x_{k+1,i} \leftarrow x_{k,i}$ - $\gamma_k \cdot \overline{g}_k$
6: **end for**

The straightforward implementation of Ring AllReduce incurs high inefficiency when stragglers appear. The goal of RNA is to propose a communication primitive that can balance the efficiency between the system and algorithm. When the collective operation is initiated, the synchronization procedure involves updating the weights of contribution from each process. We use $w_{k,i} = 1$ to indicate that the process $k$ at iteration $i$ has gradients to be applied, otherwise, $w_{k,i} = 0$. Then the weight for each process is $W = \frac{1}{\sum w_{k,i}}$. Algorithm 5 illustrates the procedure of RNA. Specifically, according to the Linear Scaling Rule [74], RNA dynamically adjusts learning rate $\gamma_k = \sum w_{k,i} \cdot \gamma$ at each iteration. All other hyper-parameters (weight decay,etc.) are kept unchanged. The implementation of RNA can still leverage the benefit from Ring AllReduce that it can update the weights among processes in $\mathcal{O}(M)$ time because it does not change the communication graph. RNA still follows the generalization of the conventional Ring AllReduce in deep learning training among all processes.

Figure 5.1 summarizes the working examples of RNA. RNA employs two threads to execute computation and communication. In our implementation, computation is done by GPU, while gradients synchronization is by CPU, i.e., MPI. In these examples, we assume that the process $w_0$ is always selected as the initiator. At iteration $t$, suppose that $w_1$ is slower than $w_0$ and $w_1$ has no available gradients when $w_0$ completes propagation. $w_1$ initiates the AllReduce without waiting for $w_1$. Since $w_1$ contributes a $g_{null}$ gradient at this time, RNA adjusts the weight $W$ and updates parameters correspondingly. At iteration $t+1$, when $w_0$ triggers AllReduce operation since $w_1$ catches up with $w_0$ and has two gradients $g_{t+1,1}$ and $g_{t+2,1}$ available on the communication thread, the accumulated gradients are locally reduced and participate in the collective operations then. We should notice that $g_{t+2,1}$ is updated using the new parameters $x_{t+1}$ while the gradients $g_{t+1}$ uses stale parameter $x_t$. At iteration $t+2$, $w_1$ is faster than the initiator. It does not wait for the completion of $w_0$ and continue the next iteration. When the initiator $w_0$ is ready, then AllReduce is

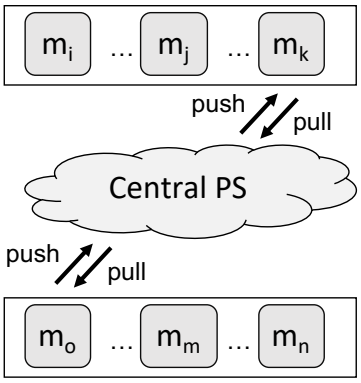



Figure 5.2: Hierarchical synchronization scheme: $m_i$ represents the $i$-th worker.

performed to update the parameters using $g^0_{t+2}$ and $g^1_{t+2}$.

While in some extreme situations, the staleness might be more than two. RNA implements a weighted averaging to reduce the accumulated locally. For work $i$ at iteration $k$, the locally reduced gradient is $g' = \frac{\sum[t-(k-\tau)+1]\cdot g_t}{\sum[t-(k-\tau)+1]}$, in which $g_t$ is the gradients obtained at iteration $t$ and $\tau$ is the largest iteration gap among accumulated results. The weight of an update is linearly associated with its iteration. If some slower processes fall behind others severely, RNA follows the design of bounded staleness [64] to overwrite the stale data and only keep results within the bound.

## 5.2 Hierarchical Synchronization in Heterogeneous Cluster

The objective of RNA is to leverage the randomized initiator to avoid the "long-tail effects." However, when this implementation is extended to a large and heterogeneous environment, the deterministic heterogeneity from hardware cannot be negligible. The mechanisms proposed so far are mainly effective in a homogeneous execution environment but do not help with slowdown situations. Slow workers who always fall behind others can enlarge the iteration gaps among workers gradually, resulting in lower accuracy. The best solution is to allow asynchronous synchronization. To achieve that, we combine the decentralized design with the traditional PS implementation, which can be illustrated in Figure 5.2. The hierarchical AllReduce first groups $M$ machines into $N$ groups, and uses three phases to do update parameters: firstly,

each group executes AllReduce operation and updates parameter among the assigned machines in this group following the basic design of RNA. This procedure follows the basic randomized Non-blocking AllReduce; secondly, the averaged gradients among each group is applied to update models using parameter server. The updated parameters from each group is pushed to a central PS from the selected initiator to be averaged, the results are then pulled back to the initiator worker; thirdly, the selected initiator in each iteration executes a broadcast operation within the group to propagate the final result to every process. In this mode, each group can be regarded as a "node" in the traditional PS. In a large scale, it is easy to implement asynchronous synchronization because each group communicates with parameter servers directly and computes gradients independently.

Whether the hierarchical synchronization should be used or not, it depends on both the system performance and application behaviors. To determine whether to choose one or more AllReduce groups, we test a simple condition of $\zeta > \upsilon$, where $\zeta$ denotes the difference of the time between the fastest task and the slowest one, and $\upsilon$ is the average time of one iteration of all processes. If $\zeta > \upsilon$, we use a two-group configuration. During the group configuration, processes are ranked according to the processing time. The processes with processing time larger than $\upsilon$ are regarded as a slower worker. Faster workers are defined in a similar way. Faster and slower workers are partitioned into two subsets. The partition-and-group procedures are recursively performed in each subset until $\zeta \leq \upsilon$ is satisfied inside the group.

It is worth noting that the proposed hierarchical synchronization is different from hierarchical AllReduce [96], which is mathematically equivalent to All-Reduce among all workers with acceleration brought by the hierarchical architecture. For RNA, workers end up with different weights after the synchronization for a different group. Compared with the default Ring AllReduce, the deterministic slowdown is avoided, making each group homogeneous. Compared with the traditional PS, the number of

Table 5.1: Notation.

| | |
|---|---|
| $\|\|x\|\|^c$ | the $l_c$ norm of vector x |
| $E_\xi(\cdot)$ | the expectation of variable $\xi$ |
| $x_k$ | the model parameter at $k$-th iteration |
| $\xi$ | the sampled data from input |
| $f(\cdot)$ | the target function for optimizing |
| $g(\cdot)$ | the gradient function |
| $\tau_{ij}$ | the iteration gap between $i$, $j$-th machines |
| $L$ | the Lipschitzian constant |
| $\sigma^2$ | the bounded variance |
| $K$ | the total number of iterations |
| $\mathbb{B}$ | the parameter weights |
| $x^*$ | the optimal parameters |
| $\eta$ | staled iteration bound |

workers that communicate with a central server is reduced from $M$ to $N$, in which $M \gg N$. As a result, the asynchronous synchronization among groups with varied capacities can mitigate network contention.

## 5.3 Convergence Analysis

We next theoretically analyze the convergence rate of RNA. Important notation is summarized in Table5.1. Based on the weighted average gradients, we make the following assumptions for analysis.

**Assumption 1.** *For widely used stochastic gradient algorithms:*

- ***(Unbiased Gradient)**:The stochastic gradient $g(x;\xi)$ is unbiased: $E_\xi[g(x;\xi)] = \nabla f(x)$.*
- ***(Bounded Variance)**: The bounded variance of stochastic gradient can be obtained as: $E_\xi(||g(x;\xi) - \nabla f(x)||^2) \leq \sigma^2, \quad \forall x$.*
- ***(Lipschitzian Gradient)**: The gradient function $\nabla f(\cdot)$ is Lipschitzian, that is to say $||\nabla f(x) - \nabla f(y)|| \leq L||x - y||, \quad \forall x, \forall y$.*

**Assumption 2.** ***Bounded delay****: the delay for updating the gradient value among all machines is bounded, which means* $\max \tau_{ij} \leq \eta$.

**Convergence bound**: AllReduce spreads the reduced gradients. With the Non-Blocking Reduce mechanism, fast machines use the staled gradient values from slow ones to update its parameters, while slow machines will utilize gradients from future iterations results obtained from faster ones. We uniformly represent the mixed-version gradients for these two conditions as $G\left(x_{k+\tau_{kj}}, \xi_{k+\tau_{kj}}\right)$, where $\tau_{kj}$ is the iteration gap to the $j$-th machine. The positive $\tau_{kj}$ represents gradients from the fast worker and, the negative one is from a slow one. Based on two assumptions above, we obtain the following convergence bound:

**Theorem 1.** *The step length sequence* $\{\gamma_k\}_{k=1,\ldots,K}$ *in algorithm satisfies*

$$\sum_{k=1}^{K}\left(\gamma_k^2\left(\frac{L}{2}+L^2\mathbb{B}\eta\sum_{\kappa=1}^{\eta}\gamma_{k+\kappa}\right)-\frac{\gamma_k}{2\mathbb{B}}\right)\leq 0. \tag{5.1}$$

*We have the following convergence rate for the training:*

$$\begin{aligned}\sum_{k=1}^{K}\gamma_k E\|\nabla f(x_k)\|^2 \leq & \frac{2\left(f(x_1)-f(x^*)\right)}{\mathbb{B}} \\ & +\sum_{k=1}^{K}\left(\gamma_k^2 L+2L^2\mathbb{B}\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2.\end{aligned} \tag{5.2}$$

*The convergence rate is bounded, which satisfies the same convergence properties as the asynchronous parameter server approach [97].*

**Independent staleness**: With the guarantee of the convergence bound, we further analyze that the convergence rate is independent of the staled parameters $\eta$ after a sufficient number of iterations. We let the step length $\gamma_k$ as a constant value, and we can obtain the corollary:

**Theorem 2.** *The delay parameter $\eta$ is bounded by:*

$$\frac{4\mathbb{B}L(f(x_1)-f(x^*))}{\sigma^2}(\eta+1)^2 \leq K. \tag{5.3}$$

*We set the step length $\gamma_k$ to be a constant $\gamma$:*

$$\gamma = \sqrt{\frac{f(x_1)-f(x^*)}{\mathbb{B}LK\sigma^2}}. \tag{5.4}$$

*after substituting the upper bound:*

$$\gamma L + 2L^2\mathbb{B}\gamma^2\eta \leq \frac{1}{2\mathbb{B}(\eta+1)} + \frac{\eta}{2\mathbb{B}(\eta+1)^2} \tag{5.5}$$

$$= \frac{2\eta+1}{2\mathbb{B}(\eta+1)^2} = \frac{1}{2\mathbb{B}}\frac{2\eta+1}{(\eta+1)^2} \tag{5.6}$$

$$\leq \frac{1}{2\mathbb{B}} \tag{5.7}$$

*According to theorem 1, we set the step length $\gamma_k$ to be a constant $\gamma$. Then we can obtain the following convergence rate:*

$$\begin{aligned}\sum_{k=1}^{K}\gamma E\|\nabla f(x_k)\|^2 \leq \frac{2\left(f(x_1)-f(x^*)\right)}{M} + \\ \sum_{k=1}^{K}\left(\gamma^2 L + 2L^2 M\gamma\sum_{j=k-T}^{k-1}\gamma^2\right)\sigma^2\end{aligned} \tag{5.8}$$

*which is equivalent to:*

$$\frac{1}{K}\sum_{k=1}^{K}E\|\nabla f(x_k)\|^2 \leq 4\sqrt{\frac{\left(f(x_1)-f(x^*)\right)L\sigma^2}{\mathbb{B}K}}. \tag{5.9}$$

**Discussion** According to the Theorem 2, we can find that when the $K$ is large enough, $\frac{4\mathbb{B}L(f(x_1)-f(x^*))}{\sigma^2}(\eta+1)^2$ is greater than $O(\eta^2)$. We make the following conclusion regarding the bound. First, because we analyze non-convex objectives, for a

given sequence of learning rates, the algorithm will converge to a point of negligible gradient. Convergence can be achieved by this algorithm asymptotically. Second, The convergence rate can be achieved by $O(\frac{1}{\sqrt{\mathbb{B}K}})$. The maximum delay and the number of "missing" gradients per iteration can be minimized. It can be concluded that the convergence rate is guaranteed while it requires the additional performance cost of synchronization with the slower convergence. Since the communication cost is also independent of nodes according to the analysis in [26], our optimization techniques can achieve bandwidth-optimal performance with partial synchronization, which will be further empirically confirmed by evaluations in Section 5.6.

## 5.4 Implementation Details

RNA is implemented using C++11 and Python on top of Horovod. We implement the key functionality of partial AllReduce in the package *controller*, which serves as the coordinator to initiate and perform the AllReduce to average gradients among processes. *Controller* resides in the root node in the cluster, which serves as a centralized mechanism to decide the time to execute the decentralized AllReduce. A plugin is developed for TensorFlow to enable the cross-iteration feature. As for the hierarchical synchronization, we follow PS-lite [98] to implement asynchronous communication, which provides flexible and high-performance operations such as zero-copy *push* and *pull*.

**Controller.** We implement partial AllReduce using MPI only now because RNA separates communication from computation so as to avoid the resource contention for GPU, removing barriers from communication. In default, the controller records the count of received tensors at each iteration. Only if it is equal to the number of participated processes, then the synchronization is initiated. Then AllReduce is performed by Open MPI to synchronize the obtained gradients. RNA selects one process as the initiator based on the randomized algorithm for each iteration. When the initiator has tensors ready for reduction, the AllReduce is performed by the background MPI. As

for other processes, RNA will *sum* the gradients locally if there are multiple available tensors, which are from different iterations. As for stragglers, each process allocates a *null* gradient, i.e., *null* tensor in TensorFlow, as the input by default, whose size and shapes are exactly identical to each other among processes. After each AllReduce operation, the input gradients are overwritten by a *null* gradient so as to avoid using outdated gradients. When the AllReduce is completed, RNA returns a callback with the $iterationID$ via $EnqueueTensorAllreduce$. The $iterationID$ records the step of synchronization. Also, the returned output gradients overwrite the previous results on each worker. Note that all of the above input and output gradients are cached on the CPU memory. NCCL [45] can be applied to synchronize gradients among remote GPUs, which requires additional GPU memory buffer among GPUs to cache input and output gradients individually.

**TensorFlow plugin.** In default, Horovod wraps TensorFlow optimizer in DistributedOptimizer, which is an opt-in graph optimization module. It supports altering runtime behavior of graph execution, such as instrumenting OpKernel implementation, adding and removing data, or control dependencies of graph nodes. The TensorFlow plugin goes through the data-flow graph to obtain gradients before its execution at each iteration. To enable cross-iteration training, we create two TensorFlow ops. The $WriteOp$ caches the obtained gradient on the CPU memory. If there is a *null* tensor, it will be replaced by the new input. If there are input tensor waiting for reduction, it will be accumulated. To avoid being block by the default *allreduce*(), a new kernel, i.e., *ReadOp*, is created. It first checks if there is a new output tensor available, according to the $iterationID$. If yes, the new gradients are copied into the GPU memory to replace the local gradients via PCIe. Otherwise, local gradients are used to fit the deep learning model. If RNA with hierarchical synchronization, the *get_weight*() will write the parameters to CPU buffers after *apply_gradient*() is finished. Then the updated tensors are pulled back. RNA overrides the *set_weight*()

API that is inherited from Tensorflow *optimizer* to use the averaged gradients from CPU to continue training. With these two new kernels, the computation of TensorFlow does not necessarily wait for the completion of the communication.

**Hierarchical synchronization.** A parameter server (PS) is a logically separate device that stores global parameters and provides a key-value interface to workers. Generally, PS approach has the following phases: (1) each worker computes the gradients using its local sampled data and sends them to PS (push); (2) central server aggregates the gradients across workers and updates its parameters (update); (3) Workers synchronize parameters with PS (pull). Following the logic of ps-lite, a $notify\_ready$ value is returned from the callback by $get\_weight()$. The PS only executes the parameter summation, i.e., model averaging, which require additional CPU resources. Fortunately, modern CPU are good at summation operation due to the highly optimized AVX instructions [99]. The additional computation on the CPU will not be the bottleneck. Then $PSPushPull()$ is called to perform a *push* and *pull* operation on the output tensors sequentially. Only the selected initiator serves as the "node" and triggers the $PSPushPull()$ operation. We applied the default $wait()$ API to lock the variables on each worker. After the gradients are aggregated and being updated at the central server, the updated parameters will be pulled back to the initiator. The pulled back parameters are written to the CPU buffer. RNA notifies MPI to broadcast the new tensors among the group via $broadcast()$ to overwrite the output tensor with the same $iterationID$, and then the parameters are unlocked. The $set\_weight()$ API in TensorFlow uses the updated parameters for propagation. The hierarchical synchronization is executed asynchronously across all processes periodically. We leave the frequency tuning as our future work.

Table 5.2: The configuration of hardwares.

| Processor | GPU Model | Num. |
|---|---|---|
| Intel 3.2GHz Xeon E5-2667 v3 | 2 × Nvidia Tesla K80 GPUs | 4 |
| Intel 2.60GHz Xeon Silver 4112 | 8 × NVIDIA GTX-1080Ti | 2 |
| Intel 3.2GHz Xeon Bronze 3104 | 2 × Nvidia GTX-2080Ti | 4 |

## 5.5 Evaluation Setup

### 5.5.1 Testbed Setup

We use a local cluster to evaluate the performance of the proposed RNA model and implemented mechanism. Table 5.2 lists the hardware configurations of the machines in the cluster. These machines are connected with EDR Infiniband. All nodes in this cluster run Ubuntu Server 16.04 with MPI 4.0.1, Python 3.7, CUDA 10.1, cuDNN 7.6.0, gcc 8.1.0, g++ 8.1.0, TensorFlow 2.1.

As for the dynamic system heterogeneity, we follow the experiment setting as Hop [33] to inject delays to simulate the system heterogeneity. Each worker is slowed down randomly in each iteration, where $n$ is the number of workers.

### 5.5.2 Deep learning models and datasets

To evaluate the performance of RNA and compare it with other works, we train three kinds of neural network models on real datasets, including image classification, machine translation, and video processing.

#### 5.5.2.1 image classification

ResNet50 is a convolutional neural network that is 50 layers deep used for image classification. We train ResNet50 [49] model over ImageNet dataset [100], which contains 1,281,167 images to be classified into 1,000 classes. The model contains 25,559,081 parameters. Momentum optimizer is used with $momentum = 0.9$ and $weight\_decay = 5 * 10^{-5}$. The initial learning rate is 0.125 and decays to its 0.1× on

epochs 30, 60, 80. The batch size is 128.

VGG16 [10] is a communication-intensive network with thirteen convolution layers of a 3×3 filter with a stride 1. It is a pretty large network, and it has more than 138 million parameters. We train VGG16 on dataset CIFAR-10 [93], whose evaluation setup is batch size: 128, learning rate: 0.1, momentum: 0.9, weight decay: $10^{-4}$.

#### 5.5.2.2 machine translation

Transformers [32] are developed to solve the problem of neural machine translation, which transforms an input sequence to an output sequence. We train Transformer on WMT17 dataset [101], which is an English to German translation dataset. The initial learning rate is set to be 2.0. The model has 61,362,176 trainable parameters. While training the model, we use the varying input length. The samples in the training dataset typically consist of sentences in various lengths. Thus the computation overhead varies with the length of the input and output sentences, leading to unbalance training time.

#### 5.5.2.3 video processing

RNN [102] is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. LSTM is a kind of recurrent neural networks that are intimately related to input sequences and lists. We train a single, 4096-wide LSTM layer, followed by a 1024 Dense layer, with some dropout in between on UCF101 [61]. The model has 34,663,525 parameters. UCF101 has 13,320 videos from 101 action categories, which gives the largest diversity in terms of actions and with the presence of large variations in camera motion, object appearance, etc. We also use the varying input length to train the model, which is linearly associated with the length of videos. The input batch size is 128.

### 5.5.3 Approaches and Performance Metrics

We compare the performance of RNA with three other synchronization models: Horovod [28], AD-PSGD [34], and eager-SGD [86]. Horovod is selected as the state-of-the-art baseline, which significantly outperforms many other implementations of All-Reduce. To achieve better performance, NCCL is configured to achieve better AllReduce speed. The Tensor Fusion is also enabled for better network utilization. Tensor Fusion can reduce the overhead when performing AllReduce operations on gradients by avoiding frequent initialization. AD-PSGD is implemented in TensorFlow by randomly selecting communication neighbors. It uses grpc to communicate parameters between nodes. eager-SGD proposes solo and majority collective communication to implement partial AllReduce. Since in a heterogeneous environment, it happens that few processes are always faster than the others. The solo collective communication may negatively impact the convergence because of the staled update from slower processes. So we only implement the majority collective communication as the baseline.

We use the time it takes for the model to achieve the target loss as the metric of performance. We also measure the number of iterations and averaged per-iteration time to analyze the effect of our optimizations. We further use the validation dataset to validate the accuracy of the obtained models. As for Transformers, we conduct fixed-time experiments to compare the throughput of a different solution.

## 5.6 Experimental Evaluation

### 5.6.1 Training speedup and convergence

We evaluate the training speed of ResNet50, VGG16, and LSTM with different system configurations. Because ResNet50 and VGG16 are balanced workloads after being preprocessed, we introduce system delay randomly, which ranges from 0 to 50ms, on each process. To evaluate the performance of the hierarchical synchro-

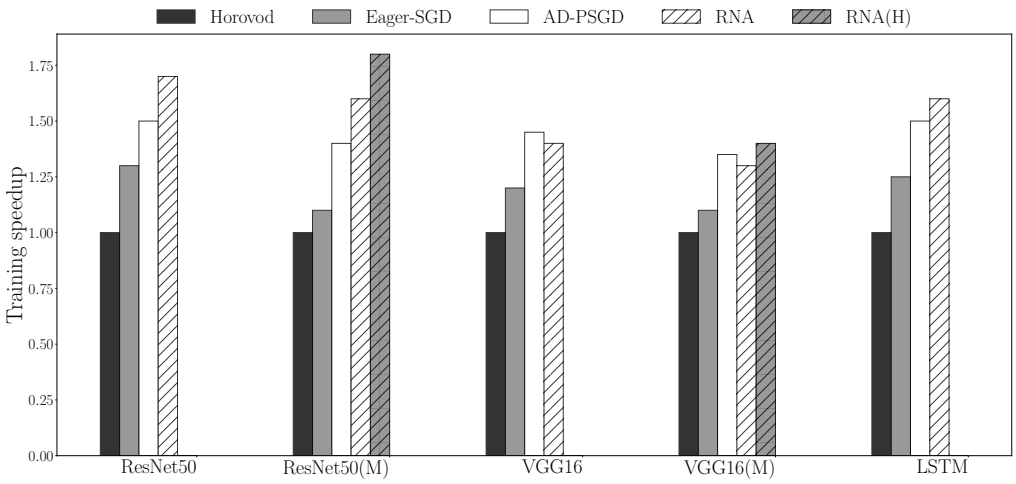



Figure 5.3: Training speedup by RNA compared to Horovod, eager-SGD, and AD-PSGD. "M" represents the mixed heterogeneity. "H" means that RNA is configured with hierarchical synchronization.

nization mechanism, we simulate a cluster with mixed heterogeneity by dividing the machines into two groups, A and B. For group B, higher system delay is injected, which ranges from 50 to 100ms randomly. Based on the design principle of hierarchical synchronization, two ring communication graphs are formed, and one central PS is created to coordinate the parameter synchronization. We train these three models on corresponding datasets. The goal of training is to minimize the loss value. We use Keras EarlyStopping to check whether the loss is no longer decreasing at the end of every epoch. The patience is set to ten, i.e., the training process is terminated if the loss cannot be decreased within ten iterations.

From Figure 5.3, it can be learned that RNA outperforms eager-SGD for ResNet50, VGG16, and LSTM. Compared with the state-of-the-art Horovod, RNA can achieve training speedup for ResNet50, VGG16, and LSTM by 1.7×, 1.4×, and 1.6×. The performance improvement demonstrates that the randomized per-process approach can mitigate the impact of dynamic heterogeneity probabilistically. Specifically, when training ResNet50 in a cluster with a higher degree of heterogeneity, the training speedup brought by eager-SGD and RNA with hierarchical synchronization are de-

Table 5.3: The final training accuracy for different neural networks. ResNet and VGG represent ResNet50 and VGG16, respectively.

| approaches | neural networks | | | | |
|---|---|---|---|---|---|
| | ResNet | ResNet(H) | VGG | VGG(H) | LSTM |
| Horovod | 78% | 79% | 93.4% | 93.2% | 88.2% |
| eager-SGD | 76.2% | 75.8% | 92.8% | 92.2% | 87.5% |
| AD-PSGD | 70.8% | 68% | 86.8% | 87.6% | 78.8% |
| RNA | 78.2% | 77.8% | 92.6% | 92.4% | 87.8% |

creased, which are from 1.3× to 1.1× and from 1.7× to 1.5×, respectively. However, the RNA with hierarchical synchronization shows stable performance improvement, which is 1.8× and 1.4× for ResNet50 and VGG16, respectively. It demonstrates the probabilistic approach cannot handle the deterministic slowdown, i.e., group B's machines are slower than A by 50ms on average at each iteration. The hierarchical synchronization mechanism can avoid mixed heterogeneity. It can also be noticed that the performance of AD-PSGD is higher than RNA for VGG-16. It is because VGG16 has a larger neural network, which makes the communication a dominating factor. Specifically, RNA requires extra memory copy between CPU and GPU. However, from Table 5.3, we can see that AD-PSGD achieves the lowest accuracy compared with the other three approaches when the training is terminated. Horovod, eager-SGD, and RNA can achieve high accuracy.

Specifically, the convergence curve for LSTM using different approaches is shown in Figure 5.4. Though AD-PSGD reaches the stopping criteria for training earlier than Horovod, i.e., shorter execution time, it sacrifices the model accuracy. Compared with Horovod, RNA lowers the training time from 8,200 ms to 5200ms, leading to nearly 1.6× speedup. Eager-SGD can achieve similar accuracy with Horovod and RNA, but its throughput is lower than RNA, resulting in longer execution time. Overall, RNA can both speed up the training process and guarantee good model accuracy.

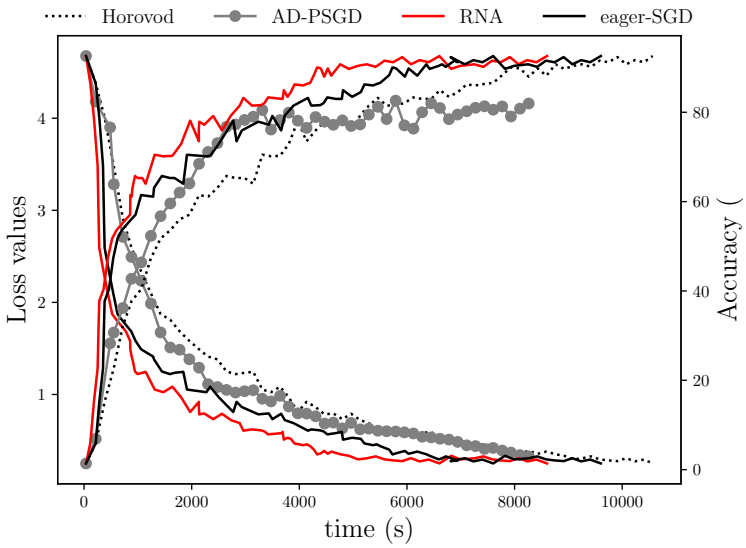



Figure 5.4: Convergence curve in terms of loss value and training accuracy for LSTM. Each point is collected at the end of one epoch.

Table 5.4: The validation accuracy for different neural networks.

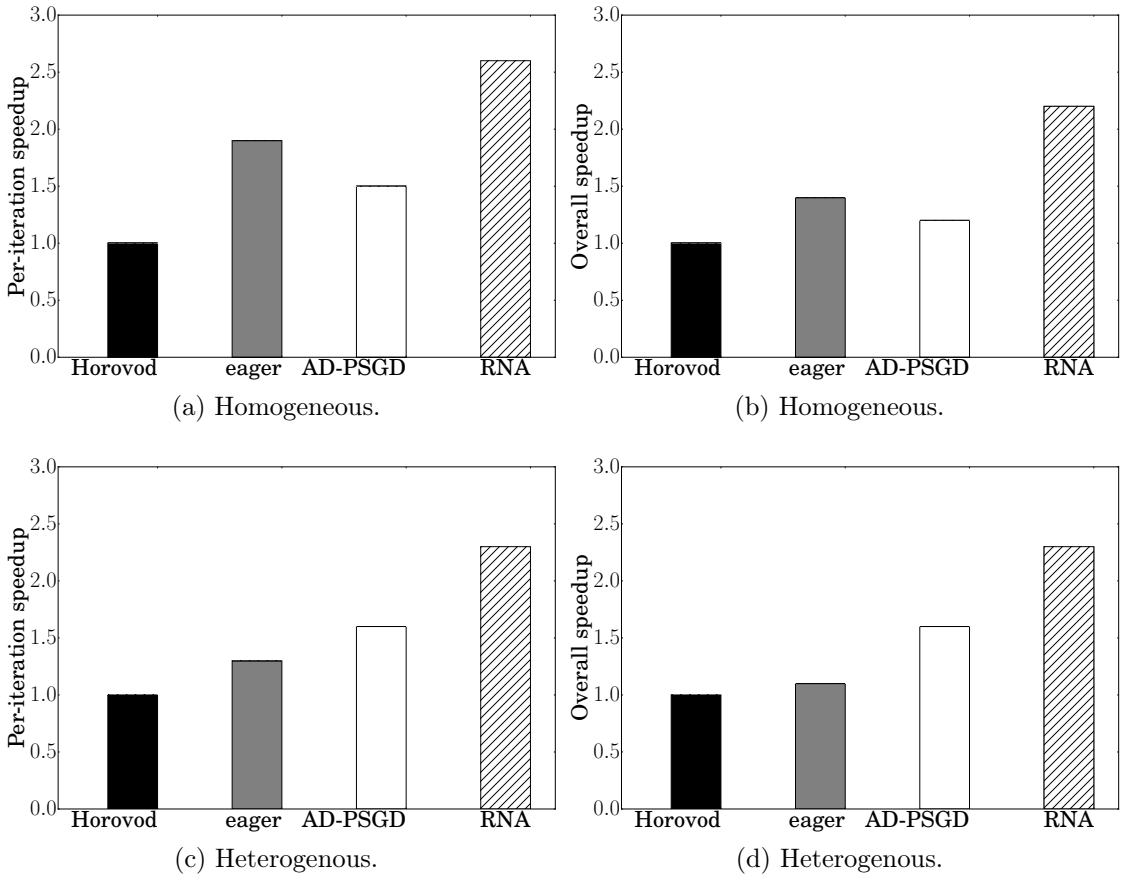
| models | approaches | # of iterations | top-1 acc. | top-5 acc. |
|---|---|---|---|---|
| ResNet50 | Horovod | 42200 | 76.2% | 93.2% |
| | eager-SGD | 49800 | 74.8 | 91.2% |
| | AD-PSGD | 38800 | 68.8 | 88.6% |
| | RNA | 52400 | 75.9% | 92.6% |
| VGG16 | Horovod | 1080 | 92.5% | - |
| | eager-SGD | 1360 | 91.8% | - |
| | AD-PSGD | 880 | 82.8% | - |
| | RNA | 1420 | 92.2% | - |
| LSTM | Horovod | 8120 | 68.2% | 94.8% |
| | eager-SGD | 9600 | 66.8% | 94.6% |
| | AD-PSGD | 7800 | 60.6% | 90.1% |
| | RNA | 9660 | 66.5% | 95.2% |

### 5.6.2 Validation on models

We further test the accuracy of the obtained ResNet50, VGG16 and LSTM models, which are summarized in Table 5.4. From the number of executed iterations, we can see that the state-of-the-art Horovod is bottlenecked by the throughput because of dynamic system heterogeneity or inherent imbalance. Although AD-PSGD requires fewer iterations to converge to minimize the loss value, the execution time of each iteration is severely affected by the synchronization overhead. And AD-PSGD achieves the lowest validation accuracy compared with other approaches. For LSTM and ResNet50, RNA takes advantage of asynchronous execution to allow more iteration in a fixed duration, leading to higher throughput. RNA can achieve higher training throughput than Eager-SGD because it can efficiently reduce the response time at each iteration. Both eager-SGD and RNA can obtain higher model accuracy than AD-PSGD. From these results, we can learn that RNA has significant convergence speed improvement compared with the state-of-the-art approach, AllReduce in Horovod. While compared with AD-PSGD, higher model accuracy is guaranteed.

### 5.6.3 Throughput comparison

To evaluate the throughput, we train Transformers in both homogeneous and heterogeneous clusters. The homogeneous cluster has two nodes configured with eight NVIDIA GeForce GTX-1080Ti for each. In the homogeneous environment, the high variance of the input sentence length incurs imbalance training time among processes. In the heterogeneous environment, we inject the additional dynamic slowdown to evaluate heterogeneity tolerance. In the experiment, we set the batch size to 4,096 tokens. The per-iteration speedup, and the overall speedup is shown in Figure 5.5. Horovod is selected as the baseline since it strictly follows the BSP model. The per-iteration time is the training time required for each iteration, which is averaged from one epoch. The overall speedup is the convergence time over the baseline. From Figure 5.5a,




(a) Homogeneous. (b) Homogeneous.

(c) Heterogenous. (d) Heterogenous.

Figure 5.5: Per-Iteration Speedup and Overall Speedup comparison among Horovod, Eager-SGD, AD-PSGD and RNA in both homogeneous and heterogeneous environment.

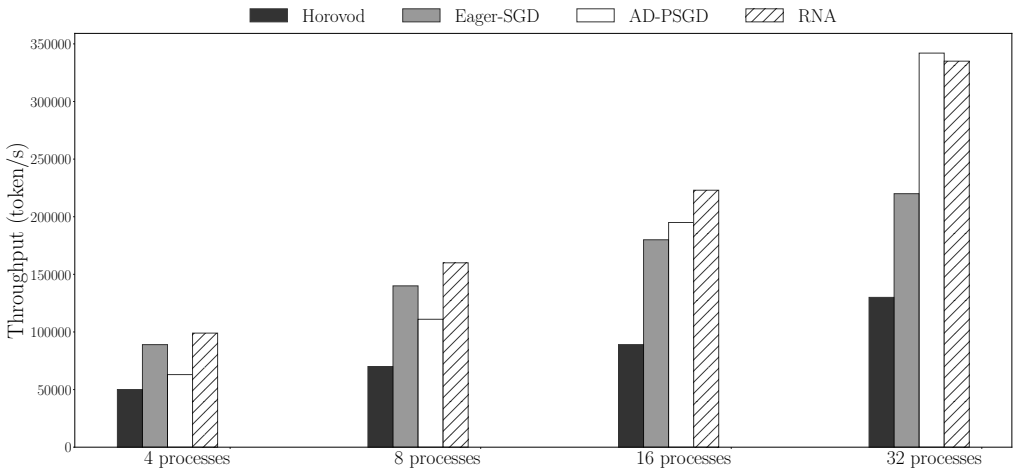



Figure 5.6: Throughput comparison among different approaches with Transformer.

we can see that RNA achieve the highest per-iteration speedup over Horovod, which is nearly 2.6× in a homogeneous environment, while eager-SGD and AD-PSGD can accomplish that by 1.9× and 1.4×, respectively. The reduction in the per iteration time results in less waiting time between iterations.More tokens have been processed by RNA within a fixed time duration compared with other approaches, leading to higher throughput. To obtain the same loss value of 2.0, RNA achieves 2.2× the overall speedup over Horovod on the execution time, as is illustrated in Figure 5.5b, while eager-SGD and AD-PSGD achieve that by 1.4× and 1.2×, respectively. In a heterogeneous environment, as is shown in Figure 5.5c and 5.5d, eager-SGD suffers from the random slowdown, whose per-iteration speedup drops from 1.9× to 1.3×. However, both AD-PSGD and RNA can achieve stable speedup, which is 1.6× and 2.3×, respectively, in terms of overall speedup. Combined with these two results, we can learn that RNA achieves a better balance between statistical efficiency and system efficiency. It requires more iterations while ignoring the staled contribution to gain significant speedup in per iteration time, leading to overall execution time speedup.

We further evaluate the scalability of RNA with other approaches on Transformers by varying the number of GPU processes. From Figure 5.6 shows that RNA and

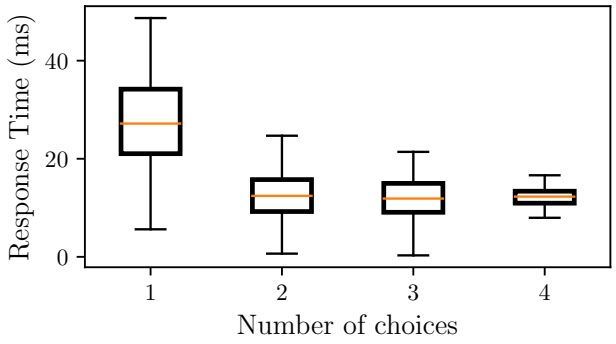



Figure 5.7: Effect of number of choices on response time. Whiskers depict 5-th and 95-th percentiles; boxes depict median, 25-th, and-75th percentiles.

eager-SGD almost achieve the highest and similar throughput on a 4-processes scale. With the increased number of processes, both the AD-PSGD and RNA achieves higher throughput than Horovod and eager-SGD. When the number of processes is increased, RNA performs better scalability than Eager-SGD and Horovod. AD-PSGD also shows superior performance in terms of scalability. In particular, we notice when the number of processes is increased to 32, AD-PSGD is has a little higher throughput than RNA. Because Transformer networks mostly consist of tensor contractions implemented as batched matrix products. As a result, Transformer requires more straightforward computation compared with the computation-intensive Convolution and causes dominated communication overhead due to the significant number of parameters, leaving less space for optimization. However, compared with AD-PSGD, we notice that RNA can reach 24 on BLEU score while AD-PSGD can only obtain 22. This result shows that RNA can ensure higher accuracy compared with AD-PSGD. In terms of throughput, RAN can achieve better scalability compared with eager-SGD. These results show that the relaxed synchronization in RNA does not sacrifice high accuracy compared to state-of-the-art solutions while ensuing the training throughput.

Table 5.5: The transmission cost in RNA.

| DL application | ResNet50 | LSTM | VGG16 | Transformers |
|---|---|---|---|---|
| Extra cost | 6.2% | 3.8% | 23% | 18% |

### 5.6.4 Sensitivity analysis

The number of choices to approximate the behavior of the system could affect performance [37]. We design a microbenchmark to evaluate the performance of the per-process sampling approach. The simulated cluster has 100 nodes. We simulate the unbalanced workload by injecting tasks to each process with randomized skewness, which ranges from 10 to 50ms. At each iteration, we randomly select a number of processes as the probes. When the fastest one among probes finishes execution, the computation proceeds to the next round. We run the synthetic workload for 100 iterations and obtain the response time for each iteration, which is shown in the Figure 5.7. The figure demonstrates that using one more oversampling probe could significantly improve performance compared to selecting initiator randomly, which reduces median response time by more than 2.4× compared to random sampling from 28ms to 12ms on average. Furthermore, the deviation of execution time for each iteration is smaller than using random selection, i.e., choice of one. The figure also demonstrates an interesting observation: a low probe ratio negatively impacts performance because it does not oversample enough to find a faster process. However, additional oversampling does not improve performance due to increased messaging. As illustrated in Section 5.3, the convergence rate trade-off more synchronization costs to gain overall execution time speedup. With this observation, to reduce the response time at each iteration efficiently, we use a probe ratio of 2 to implement our per-sampling approach.

### 5.6.5 System overhead

Compared with Horovod, to achieve asynchronous training, RNA firstly aggregates obtained gradients locally by writing the data from GPU to CPU memory. After the AllReduce operation, RNA needs to read the reduced results from CPU memory, which incurs extra transmission cost (i.e., overhead) because of the memory copy. Table 5.5 measures the transmission cost percentage in the execution time of three jobs using RNA. The transmission time for ResNet50, LSTM, VGG16 and Transformers accounts for the execution for one iteration for 6.2%, 3.8%, 23%, and 18%, respectively. We can see that the overhead for VGG16 and Transformers is more significant than that for the other jobs since these two have a larger number of parameters. Overall, the cost is much smaller compared to the performance improvement by RNA. But the transmission overhead is bottlenecked by the bandwidth of PCIe between CPU and GPU. And this communication overhead does not increase if we scale out the cluster because the transmission is executed locally. Overall, the additional transfer overhead is much smaller compared to the performance improvement brought by RNA. For neural networks with a larger model, we can optimize the performance by layer-wise overlapping between GPU and CPU.

## 5.7 Summary

This work discusses and tackles the challenging straggler problem caused by imbalanced training load in the Ring All-Reduce protocol. The imbalance can be from the dynamic system heterogeneity itself or inherent workload. I propose a new synchronization mechanism, RNA, to implement a straggler-tolerant and BSP-compatible AllReduce to improve distributed deep learning performance. The key idea is that RNA allows partial processes to synchronize their gradients without waiting for slower ones. RNA can address performance issues in AllReduce using probabilistic approach, including the straggler problem caused by dynamic system heterogeneity and asym-

metric workloads incurred by imbalance input data. Comprehensive evaluations have been performed with various DL applications in different environments while providing convergence proof for the asynchronous gradient descent algorithm. Experiment results on three representative deep learning applications, including image classification, machine translation, and video processing, show the proposed solution can achieve 1.8× speedup over the state-of-the-art implementation, i.e., Horovod, and 1.3× speedup over AD-PSGD.

# CHAPTER 6: Communication-efficient System for Training Sparse Models

## 6.1 System Design Principles

To pursue system scalability, high training throughput as well as good model quality, three design principles are proposed and implemented to build an efficient distributed training system for large-scale recommendation systems on the heterogeneous cluster:

#### 6.1.0.1 Hybrid communication with asynchrony

Sven proposes Alltoall operator to support hybrid embedding table placements, including GPU-GPU, CPU-GPU, and GPU-CPU communication patterns.

#### 6.1.0.2 Inter-batch pipeline execution

Sven introduces inter-batch pipelining to intra-batch parallelism to improve parallel training throughput further.

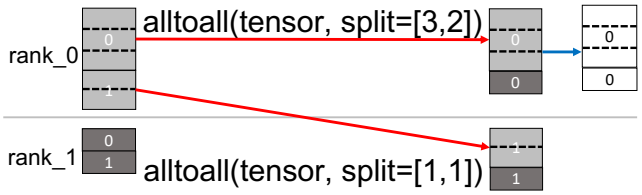
#### 6.1.0.3 Efficient embedding placement and real-time update

Sven implements a static embedding table partitioning strategy to place embedding tables considering both the resource capacity and access pattern.

### 6.1.1 Hybrid communication with asynchrony

To achieve high bandwidth and low latency over PCIe and network across processes, we introduce an asynchronous Alltoall operator for the embedding lookup results, which utilizes GPUDirect RDMA [68] as long as the destination address is on the GPU and RDMA while the destination is on the CPU. The overall workflow of Alltoall is illustrated in Figure 6.1.

To support hybrid embedding placement, an asynchronous Alltoall routine is pro-




(a) scatter-based Alltoall.

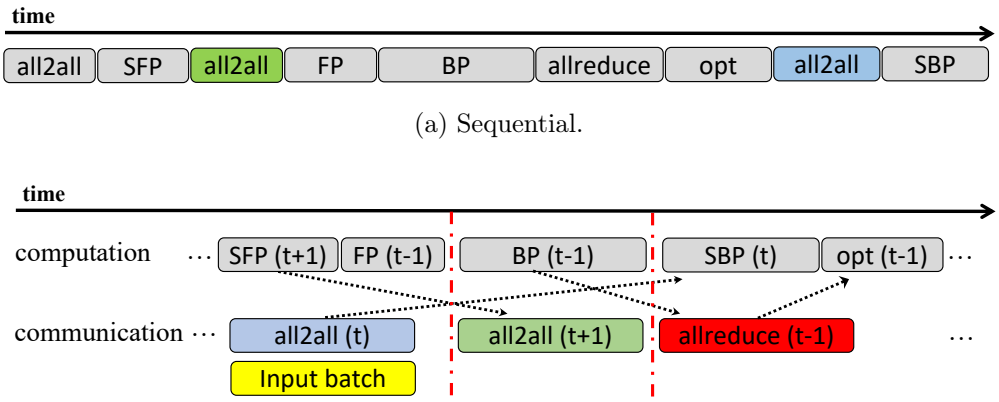
Figure 6.1: The design of alltoall API. Red lines represent the data transfer between processes, while the blue line represents the memory copy.

posed. Inspired by the design of Horovod [28], during the initialization phase, Sven starts a listening thread on each training process, which is responsible for sending/receiving tensors. Whenever there is a communication request, the listening thread will start the Alltoall operation, composed of $n$ point-to-point direct communications. In this way, we trade the cheap CPU resources for saving the expensive CUDA compute capabilities. The underlying communication is handled by the UCX library [103], which has a broad range of optimizations for achieving low-software overheads in the communication path and allows near native-level performance. Note that a well-known terminology that is typically used in the context of networking is memory "registration" [103]. Specifically, the UCX library registers the memory so that it can be assessed directly by the hardware. Then the network stack associated with an application context can typically send and receive data from the mapped memory without CPU intervention. After memory registration, the memory handle includes all information required to access the memory locally using UCP routines. Correspondingly, a remote registration handle provides information that is necessary for remote memory access. To register memory, at the initialization stage, Sven uses the $ucp\_mem\_map()$ API provided by the UCX to "register" the GPU memory address that is being used by the training framework, which, in our case, is returned by the $VisitAlloc()$ API in Tensorflow.

NCCL Alltoall operator is blocking and synchronous, forcing each process to wait for other ranks to arrive before effectively posting the NCCL operation on the given

stream. The synchronous implementation is simple and safe, which, however, could be the performance bottleneck at a large scale. Imagine that in a distributed setting with thousands of processes, in which all ranks finish calculation at the different timestamps but have to wait for the slowest one and execute the communication simultaneously. To overcome these, an asynchronous $scatter-based$ Alltoall operator is proposed. As is shown in Figure 6.1a, if one process finishes the forward pass, it will scatter the tensor to all the participated processes immediately, including itself. Since some slower processes might not arrive at the communication stage at this point, additional memory is required to save the received data temporarily. Once the slower processes arrive at the communication stage, $cudaMemcpy$ is needed to copy the received data to the destination address for the running application. In a distributed environment consisting of $n$ trainers, the Alltoall process is composed of $n$ independent *scatter* operations, each of which is initiated once the embedding lookup results or parameter updates are ready. When all the data splits are received for each rank, a $DoneCallback$ status is returned, indicating that Alltoall for the current iteration is completed. In this way, asynchrony is achieved, resulting in lower transfer latency. However, it doubles the usage of memory and uses additional memory bandwidth. We observe that the total transfer size for AlltoAll operations typically ranges from 100s to 300s MB, while each individual data size of up to 300s of KB. This message size is not the overhead for modern accelerators while it is sensitive to interconnect latency. As for the memory bandwidth, we further propose kernel fusion for the upstream operations to alleviate the bottleneck, which will be detailed in the next section.

To support hybrid embedding placement, three communication paths are required. GPU-GPU is required for GPU embedding, while CPU-GPU and GPU-CPU are required for CPU embedding in the forward and backward pass, respectively. The InfiniBand transports provided by the UCX can achieve optimal performance using RDMA for inter-node communication. To avoid additional data transfer over




(a) Sequential.

(b) Four-stage pipeline.

Figure 6.2: Sequential and four-stage pipeline execution model. SFP and SBP represent forward and backward pass for sparse parts, respectively. FP and BP represent forward and backward pass for dense parts. Opt represents optimizer update. Blocks in green represent Alltoall communication after forward pass, and blocks in blue represent Alltoall communication after backward pass. Block in the dark is to re-distribute embedding feature data. Dashed arrows represent data dependencies. $t$ represents the time interval.

PCIe, GPU-Direct RDMA is used in the GPU-GPU path, and RDMA is used in the GPU-CPU and CPU-GPU cases when the data is transferred over the network to the remote process. However, specific optimization is needed for the CPU-GPU and GPU-CPU paths for intra-node communication. As for the CPU-GPU path, $cudaMemcpyHostToDevice$ is required to copy the data from host memory to device memory. Then the data is sent to the destination process via NVLink. The data transfer over PCIe only accounts for a small fraction because it only happens in the intra-node phase, which causes minimal overhead over PCIe. But the fast feed offered by NVLink benefits the communication process. For those GPUs which have no NVLink interconnect, the data is transferred across processes via shared memory and then over PCIe. The procedure for GPU-CPU traverses the same devices, but it is in the reverse order.

### 6.1.2 Inter-batch pipeline execution

Sven introduces inter-batch pipelining execution to improve training throughput further. Note that using a single computing device, either GPU or CPU, in the system does not provide pipelining opportunities during the training process. Sven proposes to pipeline data movement for multiple mini-batches, in which way communication can be overlapped with computation, improving resource utilization. Also, these divided stages incur cross-minibatch dependencies that our asynchronous pipeline needs to handle. This section will first detail the execution model of the default training process and propose the interleaving between neighboring iterations on each device. Second, we provide an analytical model for the pipelining execution and point out the potential system overhead incurred. Lastly, we introduce how Sven bounds the statistical staleness in an asynchronous setting. For simplicity, we only illustrate the process on GPU, which is primarily similar to CPU.

The default execution model of one iteration training is illustrated in Figure 6.2a, which runs local iterations on workers sequentially. Note that each device will only handle computation for a single layer at any given time because of the graph dependencies of the neural network. One training iteration can be characterized into two main operations: computation and communication. The total runtime of the default synchronous training can be easily summarized as:

$$\begin{aligned} t_{total} = N \times (&t_{all2all} + t_{SFP} + t_{fall2all} + t_{FP} \\ &+ t_{BP} + t_{allreduce} + t_{opt} + t_{ball2all} + t_{SBP}), \end{aligned} \tag{6.1}$$

where $N$ denotes the total number of training iterations, and $t$ represents the time taken by each stage. Synchronous training depends on the summation of execution time taken by all stages, which leads to long end-to-end training time. Also, the sequential execution model causes resource under-utilization since the machine remains idle during the communication stages, for example, the Alltoall and Allre-

duce operations. Inter-batch pipeline execution injects more mini-batches to relax the iteration dependency. To keep the pipeline full and thus achieve high hardware efficiency, Pipedream [57] and GPipe [56] inject sufficient mini-batches in an epoch into the pipeline as long as there is an available resource. However, this approach could not be directly applied to recommendation models since they require several communication-intensive global synchronizations.

A four-stage pipeline execution is proposed to enable interleaving between neighboring iterations while maintaining globally synchronized communication, as is shown in Figure 6.2b. The reason why Sven divides the procedures into four stages instead of other numbers is that there are four main communication operations, including input re-distribution, two Alltoall operations for sparse parts, and AllReduce for the dense part. The first stage, which is represented as a blue bar in the Figure, happens when the updates from dense optimizer for iteration $t$ are being distributed across all processes via Alltoall. The forward pass on the sparse parts for the future iteration $t+1$ is scheduled in parallel with $t$ Alltoall to reduce execution bubbles. Since the sparse forward can be finished earlier than the Alltoall. The forward pass for the dense part using the stashing data from iteration $t-1$ is injected, fully overlapping the communication. In the second stage, which is represented as a green bar, since the results from embedding lookup for the iteration $t+1$ are ready, the corresponding Alltoall can be executed in parallel with the backward pass for iteration $t-1$, using the stashing gradients from the upstream. Then, it comes to the third stage, which is represented as the red bar in the Figure. The third stage, i.e., Allreduce, will average gradients for dense layers among all processes using the results from the backward pass at iteration $t-1$. In the meantime, Sven starts the embedding table update for iteration $t$ since the required input is ready in the earlier stage. As a result, the timing model for distributed training is resource bound, either communication or

computation bound, which can be formulated as:

$$t_{total} = N \times (max(t_{SFP} + t_{FP}, t_{fall2all}) + max(t_{BP}, t_{ball2all}) + max(t_{allreduce}, t_{SBP}) + t_{opt}). \quad (6.2)$$

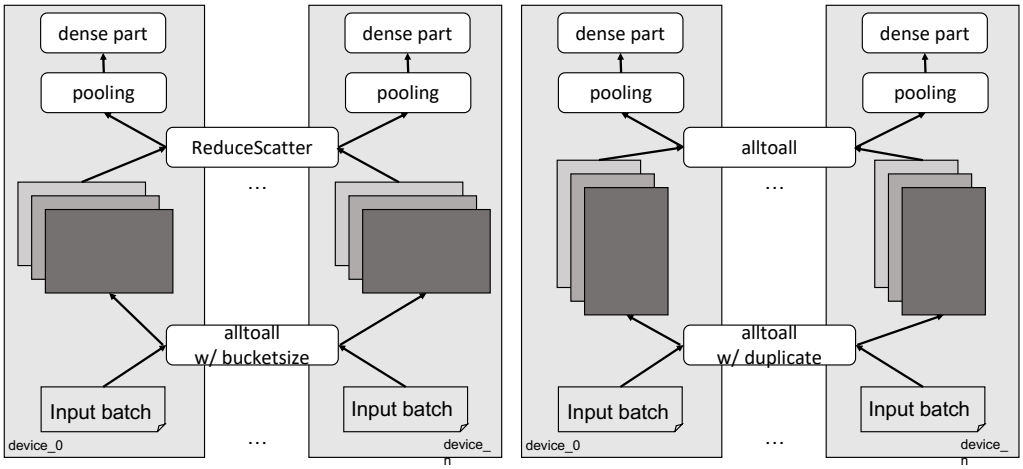
The overall execution time is reduced compared to the default execution. To ensure correctness, Sven implements a queue-based coordination scheme to support pipelining. Each stage, either computation or communication, maintains a queue based on FIFO policy. The capacity of the queue is two. For example, when the backward pass is finished, it enqueues output tensor for the downstream stage, AllReduce. The AllReduce also consumes its cache tensor based on the FIFO policy. In this way, it is guaranteed that the upper bound of staleness is under control, which is three.

As for the input re-distribution, the traditional training process requires Alltoall communication for the embedding table because the input data are pulled from the remote database in streaming batches [66], which is shown as the dark Alltoall block in Figure 6.2. Because the volumes of indices are related to the values of the lengths, the pattern of communication requires an AlltoAll operation to distribute the value lengths followed by an AlltoAll operation for input table indices. We believe the overhead from data re-distribution can be further optimized. Note that in heterogeneous GPU-CPU clusters, the GPU is responsible for the compute-intensive training task, and the CPU machines are in charge of reading and preprocessing training data into batches that the GPU can quickly use ideally with as little idle time as possible. It can be noticed that the input batch data only has downstream consumers. With pipeline execution, the data preprocessing is partitioned as an independent stage. The I/O bound data tasks, e.g., training data reading and input batch sharding, are moved to the CPU-only cluster. The processed results are transferred to the training cluster via RPC and saved in the host memory or disk. As for GPU embedding, the data transfer from host memory to device memory is required, which, however, can

be executed in parallel with the other communication operations. In this way, the Alltoall operations for input batch data are no longer needed in the training cluster, resulting in the reduction of end-to-end training time and resource contention. This stage can always be parallel with other stages, which is represented as the yellow bar in Figure 6.2b.

, However, some operations during the forward or backward passes, such as embedding lookup, pooling, etc., are memory-bound operations. The memory bandwidth issue will be more severe when pipelining computation with communication. To alleviate this issue, we further adopt kernel fusion to optimize these operators to reduce the global memory access and data loading/storing operations. For example, instead of launching a sequence of individual embedding lookup kernels for each embedding table, Sven fuses these sequential embedding lookups operations into a single CUDA kernel, improving parallelism and memory bandwidth utilization. Besides batching multiple embedding tables together, the backward pass is fused with the optimizer operator for the sparse part to further reduce the memory requirements and avoid additional memory access.

We further investigate the generated memory overhead after introducing the pipelining scheme. For the default procedure, since the execution is sequential. It only requires one replica of the intermediate data during the training. The minimum required memory space for each stage is $max(m_{comm},\ m_{comp})$, in which $m_{comm}$ and $m_{comp}$ represent the memory requirement for communication and computation for one specific stage, respectively. Sven adopts the queue-based coordination scheme to schedule the training stages. For each stage, it acts as the "producer" for the downstream tasks and the "consumer" for the upstream tasks. For any time interval, there can only be one computation and communication task being executed in parallel. Generally, we can transfer intermediate data from device memory to external resources such as host memory and then prefetch the data back when needed to reduce the




(a) Row-major. (b) Column-major.

Figure 6.3: Two different embedding table sharding schemes.

required memory space [19]. The minimum required memory space with pipelining is formulated as $2 \times (m_{comm} + m_{comp})$. The pipelining scheme makes precious memory more stringent, especially the device memory, which motivates us to propose a more efficient embedding table placement scheme.

### 6.1.3 Efficient embedding table placement

When it comes to embedding table placement, there are several challenges needed to be addressed. How to shard a group of embedding tables? How to decide the placement of sharded embedding table? How to control memory usage during long-term training? Sven proposes several strategies to tackle these challenges.

There are three popular embedding table sharding schemes: table-based, row-based, and column-based sharding. Table-wise sharding is simple but unable to handle embedding tables with large sizes. Row-wise sharding scheme partitions embedding table according to rows, as is shown in Figure 6.3a. A table-wise sharding scheme can process larger embedding tables and achieve better load balance than the table-wise one. However, it introduces high Alltoall communication overhead during the forward pass because partial results must be gathered and scattered to all other processes.

Column-based sharding scheme splits the embedding table based on the dimension of the embedding table, and each process handles smaller embedding dimensions, as is shown in Figure 6.3b. This sharding scheme achieves more fine-grained partitioning. The traditional training framework introduces an additional payload since it requires additional Alltoall to duplicate the input indices of the partitioned tabled across processes. However, this issue can be alleviated with the pipelined execution in Sven. The sharding process is executed in the remote CPU-only cluster, avoiding the expensive Alltoall operations in the training cluster.

How to place the embedding table is another challenge. Device memory has high memory bandwidth but limited capacity, while host memory has lower memory bandwidth but higher capacity. Access patterns to embedding tables follow the power-law distribution. Based on this observation, we propose that the relatively small collection of hot pages suggests moderate effectiveness of static partitioning strategies, where hot pages can be stored in expensive but high-performance device memory. We extend the embedding table to include meta-information such as the access frequency of each item in the embedding table. With this information, we further implement a static partitioning technique that partitions embedding tables such that frequently accessed embeddings are placed on the device memory. In contrast, infrequently used embeddings are placed on the host memory. The availability of hardware resources decides the ratio $\alpha$ between CPU and GPU placement. Specifically, $\alpha = \frac{m_G}{m_G+m_H}$ denotes the top $\alpha$ frequently accessed embedding features are placement on device memory, in which $m_G$ and $m_H$ represents the capacity configured for storing embedding tables on device and host memory, respectively. We believe this strategy is a viable solution since there exist relatively few highly accessed embeddings.

To control memory usage during long-term training, and entry replacement algorithm to maintain active features during the training process is required since the memory capacity is limited. Traditional memory cache such as LRU can maximize

the cache hit ratio. LRU discards the least recently used items first. Inspired by the design of subsampling techniques [104], Sven updates the importance weight for each embedding feature during the training:

$$w_s^{t+1} = (1 - \beta)w_s^t + \beta(p * k + n * l), \tag{6.3}$$

in which $\beta$ is the decay factor, $k$ and $l$ is the importance factor for positive and negative sample, respectively. $p$ and $n$ are the number of positive and negative samples, respectively. Different weights are assigned to positive and negative examples because positive examples (clicks) are relatively rare, while simple statistical calculations indicate that clicks are relatively more valuable in learning recommendation models. With this, Sven can score each embedding feature and decide the sequence of feature to be evicted if it reaches the memory limit.

## 6.2 Methodology and Implementation

### 6.2.1 Implementation

Sven is implemented on top of TensorFlow to utilize the system benefit and latest features from the community of TensorFlow. It can also be easily extended to other Frameworks such as PyTorch and MXNet by developing specific extensions. The plugin is implemented as the extended operator of Tensorflow. which is composed of low-level, high-performance C++ APIs. The Alltoall operator uses ucx [103] as the backend to implement GPUDirect-RDMA enabled send/receive. For each training process, an individual UCX context is created to handle communication. The pipeline execution divides execution into four stages. Sven introduces four new queue-based operators, which consume the tasks from the upstream stage and produce tasks for the downstream stage with a capacity of two. The newly introduced TensorFlow plugin implements function hooks and proxies for variables to schedule worker's execution patterns, which is composed of multiple high-level Python APIs. The embedding

Table 6.1: Datasets for evaluation.

| datasets | feature IDs | samples | parameters |
|---|---|---|---|
| Criteo Ad | 33M | 45M | 0.5B |
| Avazu CTR | 49M | 40M | 0.8B |
| MovieLens | 0.3M | 25M | 2M |

placement is a profiling-based mechanism, which consists of two stages. It first runs multiple training iterations and records the feature access information for each feature index, which is called the warm-up stage. During the steady stage, the most accessed features are prioritized to be distributed across GPUs based on a column-wise sharding scheme.

### 6.2.2 Neural recommendation models and datasets

Further evaluation is performed with various public datasets to validate the performance of Sven. The Criteo Ad training dataset consists of a portion of Criteo's traffic over 24 days. Avazu CTR contains ten days of click-through data used to predict whether a mobile ad will be clicked. MovieLens is a movie rating dataset. Table 6.1 summarizes the characteristics of different datasets. We evaluate three different neural recommendation models, including Wide&Deep, DeepFM, and DCN with Sven and Tensorflow on different datasets. Wide&Deep [105] is proposed to combine wide and deep models so that wide linear models can utilize the interactions relationships among sparse features, while deep neural networks can generalize to previously unseen feature interactions. DeepFM [106] is a Factorization-Machine-based recommendation model for CTR prediction, which trains a deep component and FM component jointly. DCN [107] is proposed to handle a large set of sparse and dense features and learn explicit cross features together with traditional deep representations.

Table 6.2: Hardware details of DGX-A100.

| | |
|---|---|
| Accelerators | 8 NVIDIA A100 |
| Accelerator memory (%) | 40GB |
| System memory (%) | 2TB |
| CPU (%) | 2 socket CPU |
| Interconnect (%) | 4X Infiniband 200 Gbps |

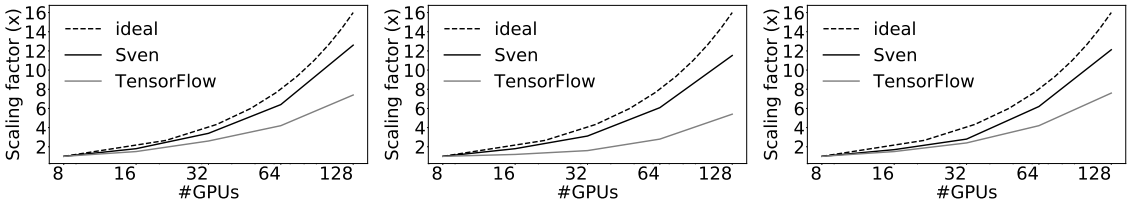
### 6.2.3 Evaluation setup

Table 6.2 summarizes the hardware configurations for each physical node. Specifically, each host is configured with dual-socket AMD EPYC 7742 64-Core Processors and 8 NVIDIA A100 GPUs with fully connected using NvSwitch and 4 Infiniband NICs to support direct RDMA among devices from different nodes. We use HugeCTR as the baseline to evaluate the performance and scalability of the proposed Sven. The CPU-only cluster consists of 28 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM. These nodes are connected with Ethernet.

## 6.3 Evaluation

### 6.3.1 End-to-end training

We evaluate the end-to-end performance of Sven in terms of scalability, throughput, and model quality. The input batch size is configured as 1024 for the best accuracy. TensorFlow [58] is used as the baseline because it follows the Parameter-server design. We follow [44] to configure the parameter server and enable the hierarchical storage for embedding tables. TensorFlow enables asynchronous parallel training, which will not enforce synchronization among processes after each iteration's update. We vary the number of nodes from 1 to 16. The input batch size is set to the same, while the size of embedding tables on each node is reduced when adjusting the number of nodes. However, the embedding tables are model-parallel, and each rank process




(a) DeepFM (b) Wide&Deep (c) DCN

Figure 6.4: Training throughput scaling for different models between Sven and TensorFlow.

Table 6.3: Training throughput (QPS).

| framework | DeepFM | W&D | DCN |
|---|---|---|---|
| TensorFlow | 5.76M | 2.49M | 1.93M |
| Sven | 2.80M | 1.14M | 1.08M |

a global mini-batch for its local table. To evaluate the performance with a smaller number of nodes, we shrink the embedding table cardinality and hash the input to be under the threshold. Since the modified embedding table negatively impacts the performance characteristics, we only study the scaling performance when varying the number of nodes. We compare the end-to-end training throughput and model quality at the 16-node scale.

Figures 6.4 report the scaling factor for both TensorFlow and Sven. The scaling factor [108] is defined as:

$$scaling\_factor = \frac{T_n}{nT_1}, \tag{6.4}$$

in which the training speed with n workers is $T_n$. The ideal performance for $n$ nodes should be $n \times T_1$, which assumes no communication overhead occurred. When increasing the number of processes, the gaps between the ideal and actual performance become larger. It can be learned that when the number of devices is increased to 128, the scaling factor is around 82% for DeepFM and DCN. But the scaling factor drops to around 68% for Wide&Deep. The achievable scaling factor for Sven is bottlenecked

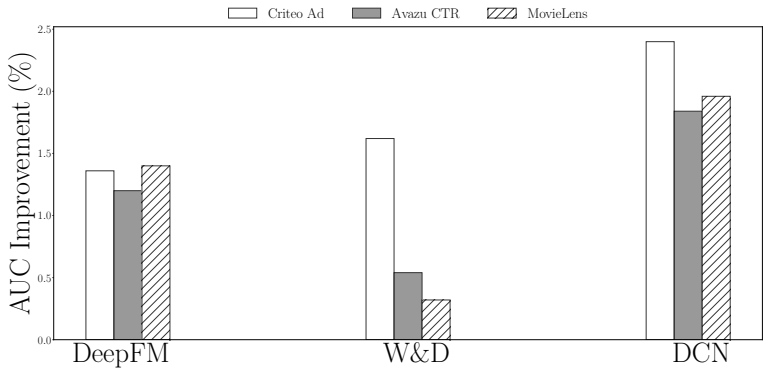



Figure 6.5: Model quality improvement achieved by Sven for different neural networks with different datasets.

by AlltoAll operations for the sparse parts, which introduces critical system overheads for both forward and backward passes. With Wide&Deep, increasing the number of processes leads to lower scaling efficiency because higher Alltoall costs occur for synchronizing the large and wide embedding table. Compared to TensorFlow, which can only achieve as high as 33% scaling efficiency at a 16-node scale, Sven shows its advantages on the training scalability. Tables 6.3 also record the obtained training throughput in terms of query per second (QPS). For the communication-intensive Wide&Deep, Sven is able to achieve 2.49M QPS, which is a 2X speedup compared to TensorFlow.

We also compare the model quality using Sven and TensorFlow regarding the area under the curve (AUC). From Figure 6.5, it can be learned that Sven outperforms TensorFlow by 0.32% to 2.4% in terms of AUC. The improvement of Sven mainly comes from the bounded staleness. Compared to TensorFlow, which does not need any synchronization across stages, Sven combines synchronous and asynchronous training via pipelining execution with the best effort to put the staleness under control.

### 6.3.2 Efficiency of communication

To further analyze the performance breakdown, we perform the evaluation on the efficiency of the implemented Alltoall operator with NVIDIA NCCL Alltoall [45] and

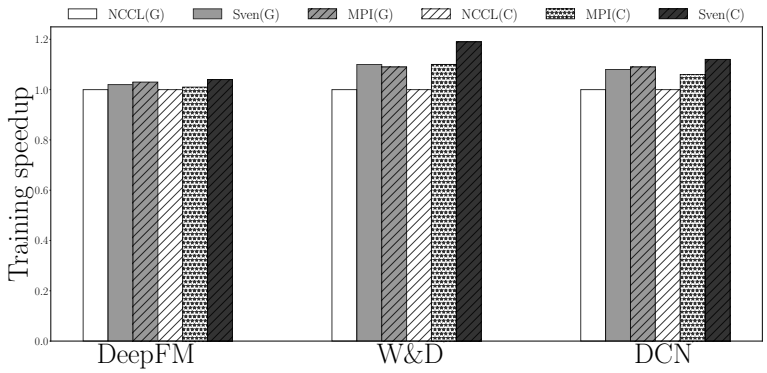



Figure 6.6: Normalized speedup compared to NCCL and MPI.

MPI_IAlltoall [109]. MPI_IAlltoall is the non-blocking version of MPI_Alltoall, which provides the non-blocking primitives that are used to achieve communication and computation overlap. We evaluate the 8-nodes scale and use fixed input data to avoid the potential data preprocessing impact. We run the same number of training iterations for each experiment and compare the latency to obtain the normalized speedup. We perform an evaluation on two kinds of embedding schemes, including GPU embedding and mixed embedding, in which GPU embedding does not require data transfer over PCIe compared to mixed embedding.

We first monitor the RDMA network throughput for all of these experiments. For both NCCL and Sven, the peak network throughput can be as high as 180Gbps and 176Gbps per network interface card (NIC), which achieves 90% and 88% network utilization, respectively. These data state that the proposed Alltoall operator can fully utilize network bandwidth as NVIDIA NCCL. From Figure 6.6, for GPU-only embedding, it can be learned that Sven can achieve speedup over NCCL by 1.1× and 1.08× for W&D and DCN, respectively. Both Sven and NCCL Alltoall can fully utilize network bandwidth, but Sven achieves lower latency. The significant performance benefit is from the asynchronous feature of the Sven Alltoall operator. Compared to MPI_IAlltoall, Sven achieves similar performance in terms of latency, which illustrates that Sven can maximize the overlap of communication and computation using

GPU-Direct RDMA. We analyze the execution logs and notice that the arrival times of the communication stage are quite different among processes. The asynchrony of the Sven Alltoall operator results in fewer bubbles on the execution timeline compared to NCCL.

In the scenario of mixed embedding, where both GPU and CPU embedding are enabled, it can be learned that Sven can achieve speedup over NCCL by 1.19× and 1.12× for W&D and DCN, respectively. The highest performance brought by Sven can be nearly 20% in terms of communication. As we have mentioned, for both GPU and mixed embedding, the network bandwidth is fully utilized with the help of GPU-Direct RDMA. However, the performance of the NCCL Alltoall operator is further decreased when involving CPU embedding. This is because NCCL Alltoall only supports input data located on the device memory. When TensorFlow detects that the input is on the host memory, it will automatically invoke the memory copy operation from host to device, introducing additional overhead over PCIe and increasing end-to-end delay. The performance improvement for DeepFM is not significant because the communication is not the bottleneck, whose input data size is around 100MB. Compared to MPI_IAlltoall, Sven can achieve nearly 8% 10% performance improvement. The improvement is higher than only GPU embedding placement since it still requires memory copy between host and device via PCIe at the destination side. Also, the overlap potential hugely degrades as the message goes larger and the scale goes higher because of the limited number of outstanding tags [110]. For the above, it can be concluded that the proposed communication components are more efficient in terms of resources usage and transfer delay.

### 6.3.3 Efficiency of pipelining

We further perform evaluations on the proposed pipeline scheme. The evaluated schemes include no-pipeline, 1-stage, and 4-stage pipelines. Specifically, the no-pipeline scheme represents that all the pipelining functionalities are disabled, in

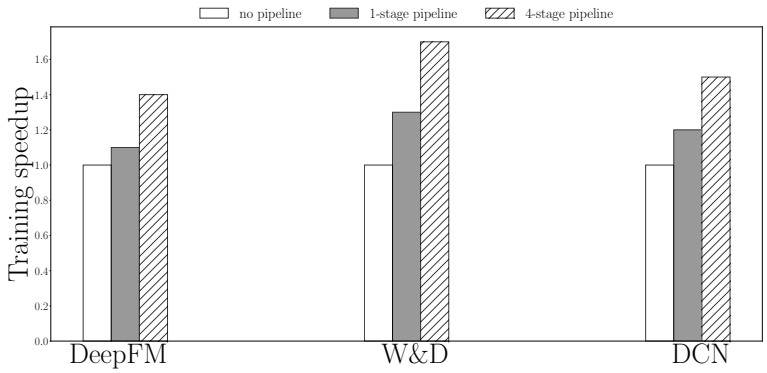



Figure 6.7: Normalized speedup comparisons between different pipeline schemes.

which the execution follows the sequential order. As for 1-stage, the input streaming stage on Kafka [111] is partitioned and moved to the CPU-only cluster. Because the input batch data has only one downstream "consumer," it can distribute the pre-processed data to a specific process based on the feature index and cache them in the queue as long as there is enough compute and memory capacity. Compared to the no-pipeline scheme, the Alltoall operation for input re-distribution is no longer needed for the training cluster. The 4-stage pipelining scheme enables all the proposed functionalities. The batch size is configured to 1024 for these experiments.

Figure 6.7 demonstrates that a 1-stage pipeline can bring nearly 11%, 21%, and 16% reduction on the training latency for DeepFM, W&D, and DCN, respectively. Hiding the long data fetch latency with the actual model training by a multi-threaded data reader is a typical solution, which has been employed by other framework [89]. However, it still requires an additional Alltoall operation to distribute the collected data records to the multiple GPUs, introducing overhead over network and I/O, especially when the CPU embedding is enabled. Sven pipelines the data preprocessing and utilizes cheap CPU-only machines to further speed up the end-to-end training. 4-stage pipeline improves the training speedup further compared to the no-pipeline scheme, which is 1.4×, 1.7× and 1.5× for DeepFM, W&D, and DCN, respectively.

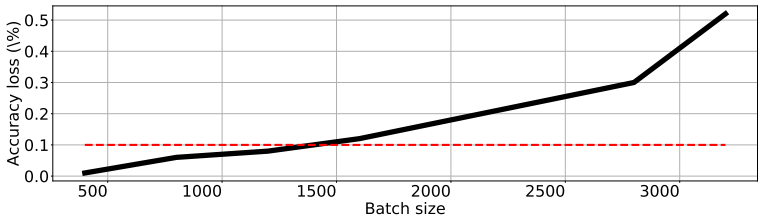



Figure 6.8: The accuracy loss with varying batch size on W&D.

The brought performance improvement shows the advantage of the 4-stage pipeline, in which the communication time spent on AllReduce and Alltoall is overlapped with computation. The results also illustrate that the pipelining scheme benefits the communication-intensive application most, e.g., the Wide&Deep model.

We also study the impact of pipelining on the model quality because it introduces bounded staleness during the training. We vary the input batch size for training Wide&Deep and record the final training accuracy loss, which is presented in Figure 6.8. Model loss regression over 0.1% might not be considered tolerable for recommendation models, which requires very well-calibrated predictions [41]. In this experiment, we use synchronous training as the baseline. The accuracy gap compared with the synchronous scheme increase with the batch size. After introducing the pipelining scheme, each stage use delayed update from previous iterations. More delayed data is involved when the batch size is increased. To best trade-off the performance and model quality, the optimal batch size should be configured as 1024, whose accuracy loss is about 0.11%. There is further exploration for better model quality using hyper-parameter tuning, which is left as future work.

### 6.3.4 Efficiency of embedding placement

In this section, we evaluate the efficiency of the proposed placement strategy. We compare the hot-based embedding feature placement to a random placement approach. As for random placement, we randomly select $\alpha = \frac{m_G}{m_G+m_H}$ faction of the total length of embedding features and store them on the device memory. We com-

Table 6.4: Training throughput on one rank (mini-batch/s).

| approaches | DeepFM | W&D | DCN |
|---|---|---|---|
| hot-based | 44 | 18.6 | 14.8 |
| random | 36 | 15.8 | 12.2 |

pare the training throughput (number of processed mini-batch per second) between these two strategies.

Table 6.4 shows that the proposed hot-based embedding feature placement can improve the training from 36 to 44 mini-batch/s for DeepFM. These results further verify that the access pattern for embedding features follows the power-law distribution. The hot-based placement strategy can benefit the training process in two aspects: first, the memory-bound operations such as embedding lookup, pooling, etc., can utilize high bandwidth memory of GPU since hot-spot data is on the device; second, the amount of data transfer over PCIe is reduced from these memory-bound operations.

## 6.4 Summary

This work optimizes the design for distributed sparse model training on the heterogeneous cluster. First, I design a new collective communication operator to support hybrid embedding table placements on heterogeneous resources. Furthermore, I propose a more fine-grained pipeline execution scheme to further improve parallel training throughput by overlapping the communication with computation and implementing an efficient partition approach to place embedding tables. Extensive experiments on different datasets with various models show that Sven can achieve as high as 2× end-to-end training speedup compared to TensorFlow.

# CHAPTER 7: Conclusions and Future Works

## 7.1 Conclusions

Deep neural networks (DNNs) have been widely applied in the field of artificial intelligence, e.g., natural language processing, computer vision, etc. There is a trend to move the deep learning training process to the heterogeneous cluster, which exhibits many unique and complicated challenges. First, as the networks go wider and deeper, the limited GPU memory becomes a significant bottleneck, restricting the size of networks to be trained. In the training of DNNs, the intermediate layer outputs are the major contributors to the memory footprint. Offloading and prefetching feature maps is one of the crucial techniques to overcome the GPU memory shortage by utilizing the CPU DRAM as an external buffer for the GPU. However, the layer-by-layer asynchronous approach cannot be effectively applied to the overlap between communication and computation, particularly for nonlinear networks. Furthermore, the default memory management policy could cause high GPU memory fragmentation for the networks with complex nonlinearities. Based on these observations, we adopt an efficient graph analysis and exploit the layered dependency structures to improve the overlap ratio. Second, decentralized algorithms, e.g., AllReduce, have been widely applied as the synchronization strategy for data-parallel distributed deep learning due to their superior performance over centralized ones. The synchronous Stochastic Gradient Descent (SGD) approach guarantees accuracy for various deep learning models, but its performance suffers from stragglers, i.e., "long-tail effects." The straggler can be caused by the inherent load imbalance from workloads or system heterogeneity. Despite existing optimizations to support centralized algorithms against stragglers, little effort has been explored in decentralized training algorithms.

Third, recommendation models are widely used in industries to help companies retain customers by providing tailored suggestions specific to their needs. However, it is difficult to directly employ GPUs on training sparse models because of the limited device memory and the large volumes of data transferred over PCIe and network in a multi-node setting. Data-parallel and model parallel schemes are combined to train the highly sparse models to overcome the memory shortage issue. Previous research in training sparse models primarily focuses on either CPU-only embedding or GPU-only embedding, thus missing the opportunities to explore more potential designs that can achieve better training throughput and lower end-to-end latency.

To support training wider and deeper neural networks on GPUs with limited device memory, a new GPU memory management runtime is proposed. This work adopts an efficient graph analysis and exploits the layered dependency structures to improve the overlap ratio. To achieve minimal memory fragmentation, we design a Group Tensors By Mobility (GTBM) placement policy to allocate tensors on the proposed unified memory pool for data structures with varied data sizes and dynamic dependencies. We implement and evaluate our system, Dymem, on several linear and nonlinear networks. Compared with vDNN and SuperNeurons, our proposed approach can achieve memory cost reduction by up to 31%. The dependency-aware strategy can improve the end-to-end throughput for nonlinear networks by up to 42%.

To mitigate the straggler problem caused by load imbalance when training DNNs in the heterogeneous cluster, this thesis proposes and implements a Randomized Non-blocking AllReduce (RNA) protocol. To avoid "long-tail effects" brought by the strict barrier in the AllReduce, this work proposes a decentralized, relaxed, and randomized sampling approach to implement partial AllReduce operation. To handle heterogeneity at a large scale, this work combines the traditional Parameter Servers (PS) with AllReduce to implement a hierarchical synchronization mechanism. This work theoretically demonstrates the convergence analysis and details the system implemen-

tation. The experiment results on representative deep learning models show nearly 1.8× speedup over the state-of-the-art Horovod and 1.3× speedup over AD-PSGD on a heterogeneous cluster.

Last but not least, this thesis introduces Sven, the optimized design for distributed sparse model training on the heterogeneous cluster. First, this work designs a new collective communication operator to support hybrid embedding table placements on heterogeneous resources. Furthermore, this work proposes a more fine-grained pipeline execution scheme to improve parallel training throughput by overlapping the communication with computation and implementing an efficient partition approach to place embedding tables. Extensive experiments on different datasets with various models show the advantages of Sven in achieving higher training throughput while maintaining model quality.

## 7.2 Future Work

There are still many interesting open directions that we can continue to explore in the future. I will further expand my domain-specific knowledge and devote my efforts to improving the efficiency of machine learning applications and the utilization of hardware, including CPUs and GPUs. In a nutshell, I will better understand the behavior of different neural networks and utilize my skills to build a scalable software system for deep learning applications at a larger scale. I believe the future machine learning systems will incorporate the following elements: automated system-level optimization; higher degrees of hardware specialization to scale system capabilities; system and algorithm co-design for practical machine learning. I highlight my future research focus as follows:

### 7.2.1 Joint Optimization for Deep Learning Training

Most of the existing DL schedulers are agnostic to the throughput scalability of DL jobs. For example, users have to specify the number of required resources when

the job is submitted, which will be fixed during the training process. Preemption or opportunistic allocation allows dynamically growing or shrinking allocated resources for a running job, which is agnostic to the statistical efficiency. Furthermore, resource-adaptive scheduling automatically adjusts the number of allocated resources based on the feedback of the training process, for example, the speedup that can be achieved if more resources are allocated. Those schedulers try to optimize the resource allocation to minimize the job completion time. None of these works consider the statistical efficiency of DL training. In this work, I plan to jointly optimize the system and algorithm configuration to achieve higher training performance. The problem is much more complicated because we have to consider both the algorithm and system factors, such as the resource allocation, the learning rate and also the batch size configured for each specific deep learning job. Firstly, it requires a new term, "goodput" to measure the performance metric, which includes both the hardware efficiency and model quality. Secondly, based on the goodput, it is expected that there should be a new scheduling architecture that jointly optimizes resource allocation in terms of cluster and the model configuration such as learning rate and batch size for each specific neural network model. I expect that the proposed solution can not only reduce the training cost for wider and deeper models but can ensure the model quality. I hold the belief that my past experiences in system optimization and domain-specific knowledge in deep learning will assist me in achieving this goal.

### 7.2.2 Auto Parallelism for Heterogeneous Deep Learning Accelerators

Currently, there are two popular approaches to parallelize DL jobs: data-parallel and model-parallel approaches. Under data parallelism, a mini-batch is split up into smaller-sized batches and is replicated across multiple devices. After the forward and backward pass, the gradients are accumulated and being updated using some variant of Stochastic Gradient Descent. Model parallelism partitions the neural networks into multiple subgraphs and distributes these subgraphs across various different de-

vices. How to parallelize the training process affects the overall performance since it will introduce different communication patterns between the devices. In the future, I will explore more systematic methods of determining the optimal tensor splitting and operator partitioning among heterogeneous hardware devices. The expected solution will be composed of several key innovations. Firstly, given a DNN model and device configuration, it requires a cost model to estimate the execution and communication cost. The mode should include the inherent factors such as the execution time, memory consumption, and environmental factors such as the resource availability resource cost. Secondly, I will propose a flexible and efficient auto-parallel system responsible for searching the optimal parallelization strategy and automatically generating the low-level execution graph based on the configuration. My experience of system design and domain-specific knowledge in deep learning can aid the exploration of efficient and scalable system optimization to achieve training deep learning applications at a larger scale and apply these techniques to state-of-the-art neural networks.

## 7.3 Publications

1. **Donglin Yang**, Dazhao Cheng, Wei Rang. "Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters." *Proceedings of the 21st International Middleware Conference.* ***Middleware*** 2020.

2. **Donglin Yang**, Dazhao Cheng. "Efficient gpu memory management for nonlinear dnns." *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing.* ***HPDC*** 2020.

3. **Donglin Yang**, Wei Rang, Dazhao Cheng, Yu Wang, Jiannan Tian, and Dingwen Tao. "Elastic Executor Provisioning for Iterative Workloads on Apache Spark." *2019 IEEE International Conference on Big Data (Big Data).* IEEE, 2019.

4. **Donglin Yang**, Wei Rang, Dazhao Cheng. "Joint Optimization of MapReduce

Scheduling and Network Policy in Hierarchical Clouds." *Proceedings of the 47th International Conference on Parallel Processing.* 2018.

5. **Donglin Yang**, Wei Rang, Dazhao Cheng, Yu Wang. "Joint Optimization of MapReduce Scheduling and Network Policy in Hierarchical Data Centers." *To appear IEEE Transactions on Cloud Computing.*

6. **Donglin Yang**, Dazhao Cheng, Wei Rang, Huanghuang Liang. "Communication-efficient System for Training Recommendation Models on Heterogeneous Cluster." Under Submission.

7. Wei Rang, Donglin Yang, Zhimin Li, Dazhao Cheng. "Scalable Data Management on Hybrid Memory System for Deep Neural Network Applications." *2021 IEEE International Conference on Big Data (Big Data).* IEEE, 2021.

8. Wei Rang, Donglin Yang, Dazhao Cheng, Kun Suo, Wei Chen. "Data Life Aware Model Updating Strategy for Stream-based Online Deep Learning." *IEEE Transactions on Parallel and Distributed Systems 32.10 (2021): 2571-2581.*

9. Wei Rang, Donglin Yang, Dazhao Cheng. "Dependency-aware Tensor Scheduler for Industrial AI Applications." *IEEE Industrial Electronics Magazine (2021).*

10. Wei Rang, Donglin Yang, Dazhao Cheng. "A Shared Memory Cache Layer across Multiple Executors in Apache Spark." *2020 IEEE International Conference on Big Data (Big Data).* IEEE, 2020.

# REFERENCES


[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of IEEE CVPR*, pp. 1–9, 2015.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, pp. 2493–2537, 2011.

[4] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *Advances in neural information processing systems*, pp. 2553–2561, 2013.

[5] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[6] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proc. of IEEE ICML*, 2011.

[7] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv:1511.06435*, 2015.

[8] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[9] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural networks*, vol. 1, no. 1, pp. 17–61, 1988.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[11] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning," *ICLR 2016, Workshop track*, 2016.

[12] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, pp. 64270–64277, 2018.

[13] "Nvidia v100 tensor core gpu." https://www.nvidia.com/en-us/data-center/v100/, 2020.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proc. of IEEE ISCA*, vol. 44, pp. 243–254, 2016.

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

[16] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proc. of ACM ICS*, pp. 1–12, 2016.

[17] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv:1412.6115*, 2014.

[18] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proc. of IEEE ISCA*, pp. 776–789, 2018.

[19] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. of IEEE Micro*, pp. 1–13, IEEE, 2016.

[20] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: dynamic gpu memory management for training deep neural networks," in *Proc. of PPoPP*, pp. 41–53, 2018.

[21] D. Yang and D. Cheng, "Efficient gpu memory management for nonlinear dnns," in *Proc. HPDC*, pp. 185–196, 2020.

[22] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, pp. 1223–1231, 2012.

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

[24] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proc. of ACM Eurosys*, 2016.

[25] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proc. of USENIX ATC*, 2014.

[26] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2017.

[27] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *Proc. of IEEE INFOCOM*, 2019.

[28] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[29] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[30] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proc. of ACM Socc*, 2015.

[31] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proc. of EMNLP*, pp. 1631–1642, 2013.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[33] Q. Luo, J. Lin, Y. Zhuo, and X. Qian, "Hop: Heterogeneity-aware decentralized training," in *Proc. of ACM ASPLOS*, 2019.

[34] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," *arXiv preprint arXiv:1710.06952*, 2017.

[35] Q. Luo, J. He, Y. Zhuo, and X. Qian, "Prague: High-performance heterogeneity-aware asynchronous decentralized training," in *Proc. ACM ASPLOS*, pp. 401–416, 2020.

[36] D. Yang, W. Rang, and D. Cheng, "Mitigating stragglers in the decentralized training on heterogeneous clusters," in *Proceedings of the 21st International Middleware Conference*, pp. 386–399, 2020.

[37] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1094–1104, 2001.

[38] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *Proc. of HPCA*, pp. 488–501, IEEE, 2020.

[39] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai, "Deep interest evolution network for click-through rate prediction," in *Proc. of the AAAI*, vol. 33, pp. 5941–5948, 2019.

[40] Z. Zhao, L. Hong, L. Wei, J. Chen, A. Nath, S. Andrews, A. Kumthekar, M. Sathiamoorthy, X. Yi, and E. Chi, "Recommending what video to watch next: a multitask ranking system," in *Proc. of ACM RecSys*, pp. 43–51, 2019.

[41] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *Proc. of HPCA*, pp. 802–814, IEEE, 2021.

[42] M. Lui, Y. Yetim, Ö. Özkan, Z. Zhao, S.-Y. Tsai, C.-J. Wu, and M. Hempstead, "Understanding capacity-driven scale-out neural recommendation inference," in *Proc. of ISPASS*, pp. 162–171, IEEE, 2021.

[43] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. Ao, W. Xu, and J. Shu, "Kraken: memory-efficient continual learning for large-scale real-time recommendations," in *Proc. of SC*, pp. 1–17, IEEE, 2020.

[44] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[45] NVIDIA DEVELOPER COMMUNITY, "NVIDIA Collective Communications Library (NCCL)," 2017. https://developer.nvidia.com/nccl.

[46] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, pp. 396–404, 1990.

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

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

[49] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of IEEE CVPR*, 2016.

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

[51] "Nvidia cnmem." https://github.com/NVIDIA/cnmem, 2018.

[52] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[53] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proc. of ICML*, pp. 1337–1345, 2013.

[54] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[55] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. of ACM Eurosys*, 2019.

[56] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, pp. 103–112, 2019.

[57] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proc. of ACM SOSP*, pp. 1–15, 2019.

[58] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. of USENIX OSDI*, 2016.

[59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

[60] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[61] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[62] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, pp. 1735–1780, 1997.

[63] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, *et al.*, "Dynamic control flow in large-scale machine learning," in *Proc. of EuroSys*, pp. 1–15, 2018.

[64] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proc. of NeurIPS*, 2013.

[65] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proc. ACM SIGKDD*, pp. 1059–1068, 2018.

[66] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, *et al.*, "High-performance, distributed training of large-scale deep learning recommendation models," *arXiv preprint arXiv:2104.05158*, 2021.

[67] B. Jiang, C. Deng, H. Yi, Z. Hu, G. Zhou, Y. Zheng, S. Huang, X. Guo, D. Wang, Y. Song, *et al.*, "Xdl: an industrial deep learning framework for high-dimensional sparse data," in *Proc. of ACM DLP-KDD*, pp. 1–9, 2019.

[68] "Mellanox ofed gpudirect rdma." https://www.mellanox.com/products.

[69] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training," *arXiv preprint arXiv:1811.03619*, 2018.

[70] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," *Google Research*, 2011.

[71] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[72] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv:1701.06538*, 2017.

[73] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[74] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[75] S. Shriram, A. Garg, and P. Kulkarni, "Dynamic memory management for gpu-based training of deep neural networks," in *Proc. of IEEE IPDPS*, pp. 200–209, IEEE, 2019.

[76] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. of USENIX NSDI*, 2013.

[77] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments.," in *Proc. of OSDI*, 2008.

[78] W. Chen, J. Rao, and X. Zhou, "Addressing performance heterogeneity in mapreduce clusters with elastic tasks," in *Proc. of IEEE IPDPS*, 2017.

[79] R. Gandhi, D. Xie, and Y. C. Hu, "{PIKACHU}: How to rebalance load in optimizing mapreduce on heterogeneous clusters," in *Proc. of USENIX ATC*, 2013.

[80] S. Wang, W. Chen, X. Zhou, S.-Y. Chang, and M. Ji, "Addressing skewness in iterative ml jobs with parameter partition," in *Proc. of IEEE INFOCOM*, 2019.

[81] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proc. of ACM SoCC*, 2016.

[82] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proc. of ACM SIGMOD*, 2012.

[83] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proc. of ACM Middleware*, pp. 253–265, ACM, 2018.

[84] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[85] J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," in *Advances in neural information processing systems*, pp. 3084–3092, 2013.

[86] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in deep learning with partial collective operations," in *Proc. of ACM PPoPP*, pp. 45–61, 2020.

[87] M. Assran, N. Loizou, N. Ballas, and M. Rabbat, "Stochastic gradient push for distributed deep learning," in *International Conference on Machine Learning*, pp. 344–353, PMLR, 2019.

[88] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proc. of SC*, pp. 1–16, IEEE, 2020.

[89] "Hugectr." https://github.com/NVIDIA/HugeCTR.

[90] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv:1410.0759*, 2014.

[91] M. Gorman and P. Healy, "Supporting superpage allocation without additional hardware support," in *Proc. of ACM ISMM*, pp. 41–50, 2008.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[93] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[94] S. Di Girolamo, P. Jolivet, K. D. Underwood, and T. Hoefler, "Exploiting offload enabled network interfaces," in *Proc. of HOTI*, pp. 26–33, IEEE, 2015.

[95] R. B. Cooper, "Queueing theory," in *Proceedings of the ACM'81 conference*, pp. 119–122, 1981.

[96] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[97] Z. Wei, S. Gupta, X. Lian, and L. Ji, "Staleness-aware async-sgd for distributed deep learning," in *International Joint Conference on Artificial Intelligence*, 2016.

[98] Distributed (Deep) Machine Learning Community, "PS-lite," 2014. https://github.com/dmlc/ps-lite.

[99] C. Lomont, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, 2011.

[100] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, pp. 211–252, 2015.

[101] EMNLP 2017, "WMT17," 2017. http://www.statmt.org/wmt17/.

[102] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[103] "Unified communication x." https://github.com/openucx/ucx.

[104] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, *et al.*, "Ad click prediction: a view from the trenches," in *Proc. of ACM SIGKDD*, pp. 1222–1230, 2013.

[105] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, *et al.*, "Wide & deep learning for recommender systems," in *Proc. of ACM DLRS*, pp. 7–10, 2016.

[106] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

[107] R. Wang, B. Fu, G. Fu, and M. Wang, "Deep & cross network for ad click predictions," in *Proc. of ADKDD*, pp. 1–7, ACM, 2017.

[108] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?," in *Proceedings of the Workshop on Network Meets AI ML*, pp. 8–13, 2020.

[109] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European*

*Parallel Virtual Machine Message Passing Interface Users Group Meeting*, Springer, 2004.

[110] M. Bayatpour, S. M. Ghazimirsaeed, S. Xu, H. Subramoni, and D. K. Panda, "Design and characterization of infiniband hardware tag matching in mpi," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 101–110, IEEE, 2020.

[111] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "Kafka: The modern platform for data management and analysis in big data domain," in *Proc. of TEL-NET*, pp. 1–5, IEEE, 2017.