Towards GPU Memory Efficiency for Distributed Training at Scale

Runxiang Cheng[†], Chris Cai[‡], Selman Yilmaz[‡], Rahul Mitra[‡], Malay Bag[‡], Mrinmoy Ghosh[‡], Tianyin Xu[†]
[†]University of Illinois Urbana-Champaign [‡]Meta Platforms, Inc.

ABSTRACT

The scale of deep learning models has grown tremendously in recent years. State-of-the-art models have reached billions of parameters and terabyte-scale model sizes. Training of these models demands memory bandwidth and capacity that can only be accommodated distributively over hundreds to thousands of GPUs. However, large-scale distributed training suffers from GPU memory inefficiency, such as memory under-utilization and out-of-memory events (OOMs). There is a lack of understanding of actual GPU memory behavior of distributed training on terabyte-size models, which hinders the development of effective solutions to such inefficiency. In this paper, we present a systematic analysis of GPU memory behavior of large-scale distributed training jobs in production at Meta. Our analysis is based on offline training jobs of multi-terabyte Deep Learning Recommendation Models from one of Meta's largest production clusters. We measure GPU memory inefficiency; characterize GPU memory utilization, and provide fine-grained GPU memory usage analysis. We further show how to build on the understanding to develop a practical GPU provisioning system in production.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; • **Computing methodologies → Machine learning**.

KEYWORDS

Distributed Training, GPU Efficiency, Resource Provisioning.

ACM Reference Format:
Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. 2023. Towards GPU Memory Efficiency for Distributed Training at Scale. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620678.3624661

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624661

1 INTRODUCTION

The scale of deep learning models (referred to as DNNs) has grown tremendously in recent years. Practitioners and researchers continue to build deeper and larger models to gain better learning performance. Nowadays, deep learning models that power the growth and main revenue stream in large technology companies have billions of parameters, and terabyte-scale model sizes [1, 5, 8, 26, 75].

Training of large-scale models demands significant memory capacity and memory bandwidth, which cannot be accommodated on a single GPU device. Therefore, various distributed training techniques were developed to enable large-scale DNN training on multiple devices [27, 46, 76, 80, 81]. Modern distributed training paradigm splits the memory-intensive model component (via *model parallelism*) and replicates the compute-intensive model component (via *data parallelism*) across GPU devices. These approaches have shown their effectiveness in training large models and increasing the training throughput in practice.

Today, GPU clusters dedicated to large-scale distributed training are common in practice [1, 25, 30, 32]. However, cluster-level GPU memory management for distributed training remains inefficient. Without effective tools, engineers often over- or under-provision GPUs for a training job. Our analysis shows that, in one of the Meta's largest production training clusters, half of the jobs utilize at most 50% of provisioned GPU memory, and 9% of the failed jobs were induced by GPU out-of-memory events (OOMs). GPUs are becoming more scarce due to wide-spread learning demands [11, 22, 47]. For example, OpenAI is heavily GPU-limited at present and GPU shortage is delaying a lot of their short-term plans [22]. Improving GPU memory efficiency in large-scale training is thus important, as training scalability and efficiency are bounded by GPU memory capacity and bandwidth.

Prior work on improving training efficiency mainly focuses on workload scheduling and load balancing [2, 3, 6, 17, 18, 20, 28, 31, 35, 45, 49, 60, 84, 91, 94], GPU memory sharing [38, 43, 56, 91, 95], and model reduction [16, 37, 76, 78, 90]. However, the aforementioned studies often assume requested

amount of GPU memory to be fixed or defined a priori. In practice, there is a significant gap between the amount of memory a training job needs and the amount it requests (§5). Hence, they do not fundamentally address GPU memory inefficiency of training jobs.

A few prior studies present measurement studies of DNN memory usage [6, 19, 38, 91]. However, they are either limited to a single GPU, or only consider data parallelism. We will show in the paper that large-scale distributed training has characteristics distinct from these studied settings. For example, model parallelism has become a norm as model size scales, which changes memory usage assumed by prior work—with data parallelism, memory consumptions are the same across GPUs; however, with model parallelism, memory consumptions differ significantly across GPUs (§6).

We realize that the understanding of GPU memory behavior in large-scale distributed training falls short in the literature. Prior work analyze DNN memory usage [19, 38, 76, 78, 90, 91, 94] based on theoretical modeling, or coarse-grained profiling on small DNNs (with sizes being around 1GB) [14, 23, 82, 85]. There is no public source for researchers and practitioners to understand the actual GPU memory behavior of production distributed training for models in multi-terabyte size and are paralleled on hundreds of GPUs or more [5, 8, 52, 58, 81]. This hinders the research and development of effective GPU memory provisioning, scheduling, and sharing solutions for distributed training at scale.

In this paper, we present a systematic analysis of GPU memory behavior of production training jobs at Meta. Our goal is to fill the knowledge gap and shed light on technical endeavors toward GPU memory efficiency for large-scale distributed training. We further show how to build on the analysis to develop a practical GPU provisioning system to improve GPU memory efficiency for large-scale training jobs in production. Our work is based on Deep Learning Recommendation Model offline training jobs in one of Meta's largest production clusters. Production DLRMs are multi-terabyte in size, consume 50+% of the AI training cycles [1, 24, 51], and serve 60% of the AI inference cycles [21] at Meta.

This paper makes the following main contributions:

- **Measurement of GPU memory inefficiency (§5).** Findings 1-3 show that: half of the production training jobs utilize less than 50% of their GPU memory; 31% of the succeeded jobs had been preempted due to GPU contention, while 9% of the job failures are due to OOM. Our results imply that solutions to GPU memory inefficiency for large-scale distributed training are in eminent demand.
- **Characterization of GPU memory utilization (§6).** Findings 4-6 show characterization of production jobs under mixed parallelisms. Distinct from well-explored data-parallel jobs, memory usage varies significantly across GPUs in jobs with model parallelisms. Memory usages incurred before and in training show distinct patterns, while both constituting the total memory usage in training. We motivate a divide-and-conquer analysis on DNN memory usage, and shed light on effective solutions to GPU memory inefficiency, in large-scale distributed training.
- **Analysis of pre-training GPU memory usage (§7).** Findings 7-9 analyze different segments of GPU memory usage before training, under model parallelism, broadcasting, optimizers, and data parallelism in production. Model parallelism dominates the total pre-training usage. We explore why model parallelisms often balance computation costs while yielding unbalanced model sizes across GPUs. Our results suggest corresponding memory provisioning strategies for different segments of the pre-training usage such as using upper bounds (§7.1) or regression (§7.2).
- **Analysis of training-time GPU memory usage (§8).** Findings 10-13 analyze training-time GPU memory usage and its relations to computation cost balance, batch size, and number of GPUs in production jobs. New memory usage incurred in training varies little across GPUs in a job when the model's computation costs are balanced by parallelism techniques. Our profiling results show that training-time usage can be decomposed as the maximum of the peak memory usages in forward pass (activation), and in backward pass (gradient and autograd).
- **Practical GPU memory provisioning (§9).** We explain the limitations of existing work in distributed training including incomplete memory accounting, substantial profiling overhead or maintenance effort. Driven by the insights of our study, we develop a practical GPU memory provisioning system named AMP, and evaluate it for the most frequently trained models at Meta. AMP can save the number of GPUs by 45% per job on average. We are in the process of deploying AMP in production.

2 BACKGROUND

2.1 Deep Learning Recommendation Model

Deep learning recommendation model (DLRM) is an important class of deep learning models widely used in personalized content ranking and recommendation service [7, 10, 15, 52, 83]. A DLRM trains user data and content and predicts output, such as clickthrough rate (the probability of the user clicking a content). The input data consists of both sparse and dense features: dense features represent continuous data and sparse features represent categorical data.

Like other popular DNNs [5, 14, 23, 52, 81, 89], a DLRM is a sequence of layers, each layer is a dense, high-dimensional tensor. Figure 1 illustrates the canonical DLRM architecture [52, 79, 96]. The model parameters of a DLRM consist of embedding tables, Multi-layer Perceptrons (MLP), feature

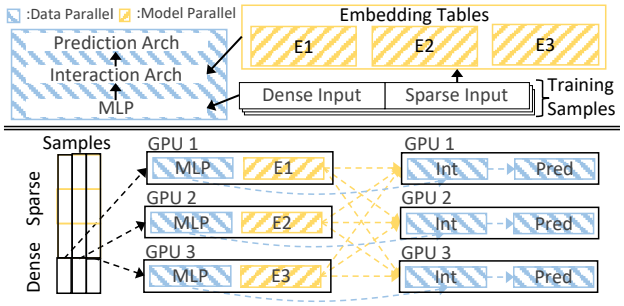

Figure 1: DLRM architecture (top), and its distributed training view (bottom) with data parallelism (blue) and model parallelism (yellow) on 3 GPU trainers.

interaction component, and prediction component. Embedding tables process sparse features into dense representation vectors, and MLP process dense features. The processed features are then fed into the interaction component to produce feature interaction representation, which is then passed into the prediction component to make prediction [52].

The dimension of a sparse feature can often reach billions [79, 96]. So, embedding tables are designed to function as lookup tables that map sparse features into dense, low-dimensional vectors, making them easier for learning. For example, one sparse feature in a training sample could be "*videos clicked in the past 7 days*". The training input of this feature could contain lookup indices to an embedding table that stores the representation vectors of all videos, where each row is a fixed-length vector of a unique video. The looked-up rows are fetched and pooled (e.g., via summation) into a low-dimensional vector that represents the value of the input feature. We refer to all the model parameters (except embedding tables) as *dense parameters*. In a production DLRM, embedding tables are often several terabytes (TBs); dense parameters are hundreds of MBs.

2.2 Distributed Training

Training terabyte-scale DLRMs under high throughput requirements demands significant memory capacity and bandwidth [1, 21, 41, 46, 79]. So, DLRM training simultaneously uses both *model parallelism* to lift the capacity constraint [80, 81] and *data parallelism* to increase throughput [12, 34, 36]. Figure 1 illustrates an example on three GPU training devices, referred to as *trainers*. Model parallelism splits the memory-intensive model component (i.e., embedding tables) across GPU trainers. Data parallelism replicates the compute-intensive model component (i.e., dense parameters) and splits training data across GPU trainers to increase throughput.

Note that the infrastructure challenges of DLRM training in terms of memory capacity and bandwidth also apply to other modern DNN training. Traditionally, DNN training often only needs data parallelism to maximize throughput as models can fit into a single GPU device. However, with the recent advent of large language models and multimodal models [4, 5, 8, 26, 58, 76, 81], terabyte-scale DNNs are becoming popular, making these challenges no longer limited to DLRM training. For example, distributed training for the recent large-scale DNNs must also use model parallelism.

2.2.1 **Training Workflow.** After a training job is configured and submitted to the cluster, it is queued for resources.

Pre-training. The job first undergoes training preparation, which we refer to as the *pre-training* phase. This phase includes model initialization, and several preparation stages:

- "Broadcast": Sharder manifests model parallelism by generating a sharding plan of how to partition the model parallel component (all embedding tables in the model) across all trainers. This stage runs a sharder [87, 88] to generate a sharding plan on the leader trainer, the sharding plan is broadcast globally to other trainers.
- "Sharding": Materializing model parallelism by loading embedding table partitions into trainers' device memory [86].
- "Optimizer": Constructing optimizers for training.
- "DDP": Initializing distributed data parallelism, e.g., replicating dense parameters, across all trainers.

These stages are essential in distributed training. They are often implemented with PyTorch libraries in communication [69], optimization [72], and data parallelism [36, 64].

Training Time. After pre-training phase, the job will start the training phase which runs the distributed training loop. A training loop repeats training iterations hundreds of thousands of times. Like other DNNs, one training iteration of a DLRM is one forward pass followed by one backward pass. Forward pass is computed from the first (input) to the last (output) layer of the model, in which model parameters are accessed to make prediction. Backward pass is computed from the last to the first layer, in which the gradients of the prediction loss with respect to (w.r.t.) the model parameters are computed and used to update the accessed parameters.

A distributed training iteration is as follows. In forward pass (shown in Figure 1), each trainer processes a different set of training samples, every sample has the same features. In each trainer, dense features is processed by the local MLP replica. Meanwhile, sparse features are sent across trainers for embedding table lookup—each trainer fetches and pools rows from local embedding tables and sends to the querying trainers, via an all-to-all communication. Then, in each trainer, the pooled vectors and MLP output are processed by its other local replicated dense parameters to output prediction. In backward pass, gradients of the loss w.r.t. the model parameters will be computed and applied to the parameters. To synchronize the replicated dense parameters, the average gradients are computed and applied to all replicas across

trainers, often through a ring-based or tree-based communication [36]. To update the embedding table, gradients w.r.t. the fetched rows will be sent to the corresponding trainer via another all-to-all communication. Each trainer uses the received gradients to update the fetched rows locally.

2.2.2 **Training Types and Model Placement.** There are two types of training: offline and online training. An offline training job trains a DLRM on large historical data. An online training job recurrently trains an offline-trained DLRM on smaller new data. In production environments, engineers aim to maximize training throughput in offline training—training a large model on as much data as possible per unit time. We focus on offline training jobs as they demand larger memory capacity and bandwidth, hence extensive GPU resources.

During DNN training, the model must reside in device memory [1, 78, 81, 91]. For DLRM, there are several options for placing the embedding tables [1, 79]. The ideal option is to place everything in GPU High Bandwidth Memory (HBM). Since most of the computations during DNN training are done on GPU, this option also eliminates CPU-GPU transfer overhead [78, 91]. Other options leverage Unified Virtual Memory (UVM)—an API from NVIDIA that provides a shared memory address space for host (CPU) and accelerator (GPU) [44]. We can either place all the embedding tables, or the less frequently accessed ones in UVM (i.e., CPU memory). Note that using UVM induces a much lower training throughput, because the memory bandwidth for fetching data from UVM is capped by the interconnect bandwidth of PCIe [46, 79], which is much smaller than HBM bandwidth [53, 55]. So, UVM options are more suitable for online training with lower throughput requirement [79]. In this paper, we focus on large-scale offline training jobs using only HBM. Nonetheless, the memory usage behavior is the same no matter the use of UVM or HBM.

3 DLRM TRAINING AT META

Models. There are hundreds of DLRMs whose offline training jobs are being actively deployed in production at Meta. They differ in model architecture, training objective, serving stage, etc. Each DLRM has an evolving version of the model as a development template. Different jobs training the same DLRM often train variants of the template model. Since many DLRMs have similar architectures, we select three representative DLRMs that have the most distinct model architecture from the top-five most frequently trained DLRMs when qualitative analysis is needed. We refer to them as Model-A, Model-B, and Model-C. Both Model-B and Model-C consist of two sub-models. The sub-models share embedding tables and MLP, with separate feature interaction and prediction components. Model-C is computationally faster than Model-B in training by design. Model-A is often the largest as it consolidates multiple DLRMs. These DLRMs predict clickthrough-rate or click-conversion-rate. Table 1 shows their production model sizes. Each DLRM has 1000+ embedding tables, with total size of 2 to 5TBs. Embedding tables occupy over 99% of the DLRM model size. But, in practice, embedding table size in a DLRM follows a long-tail distribution, e.g., 75% of them are under a couple of hundred MBs.

Table 1: Top-three DLRMs in offline training.

DLRM	Dense Parameters	Embedding Tables	
		#Tables	Size
Model-A	0.46GB	2190	5134GB
Model-B	0.80GB	1037	1988GB
Model-C	0.19GB	1864	4197GB

Table 2: GPU model specifications.

GPU Model	HBM Size	HBM Bandwidth	PCIe
NVIDIA V100	32GB	900GB/s	32GB/s
NVIDIA A100 v1	40GB	1,555GB/s	64GB/s
NVIDIA A100 v2	80GB	1,935GB/s	64GB/s

Hardware. Each machine host in the cluster is equipped with 8 GPU devices (8 trainers), and 1.5TB CPU memory. GPUs on the same host have the same GPU model. Table 2 shows the GPU model specifications.

A training job is highly configurable. ML engineers can configure the number of trainers, training hardware, and model placement options for their jobs (§2). If training hardware is not specified, the system will automatically decide on a GPU model. Each training job only uses one GPU model, i.e., no hardware heterogeneity [31, 49]. Each GPU only dedicates to one job at a time, i.e., no memory sharing due to sharing overhead [38, 43, 91, 95]. Embedding tables are queried from databases based on the model configuration during model initialization. Materialized embedding tables are stored in device memory during training. Model placement optimizations on different layers (GPU, CPU, and SSD) [79, 98] do not directly apply to our setting in this paper (§2.2.2).

Configurations. Batch size and the number of trainers are two training job configuration parameters that have dominant impacts on memory usage of distributed training jobs. They are frequently adjusted by engineers in production. Batch size specifies the number of training samples processed by the model per training iteration—a larger batch size trains the model on more data per unit time to increase throughput, and takes more memory. A larger number of trainers parallels the job on more GPUs, which can also increase throughput, by splitting samples across more trainers in data parallelism (§2). From July to September 2022, the most used batch size for Model-A, Model-B and Model-C offline training jobs is 1024 (89%), 2048 (68%) and 1024 (94%), respectively; the most used number of trainers is 128 for all

three major DLRMs (with occupancy of 90%, 72% and 88%, respectively).; among all DLRMs, the most used number of trainers is 128, and 1024 or 4096 for batch size.

Figure 2 shows the CDF of total GPU memory usage of offline training jobs to their corresponding model size, for the top-five most frequently trained DLRMs in the cluster in November 2022. The total GPU memory usage of a job is computed as the summation of peak GPU memory usage across its trainers. The model sizes of these jobs are multi-terabyte (Table 1). Figure 2 shows the total GPU memory usage in 70% of the jobs are 2-4 times larger than their model size. GPU memory inefficiency in distributed training is easily magnified at this scale.

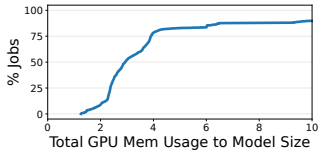

Figure 2: Ratio of total GPU mem usage to model size.

GPU Provisioning Practices. Production ML engineers have three main objectives in DNN training: (1) maximizing training throughput, (2) minimizing the number of trainers for resource efficiency, and (3) avoiding job failures (e.g., due to OOMs). Effective GPU provisioning should meet these three objectives. In this context, GPU memory is a major consideration of GPU provisioning for large-scale distributed training jobs, because training scalability and efficiency are primarily bounded by memory capacity and bandwidth [1, 3, 46, 76, 79]. However, a GPU's memory is limited.

We observe that ML engineers provision GPUs for their training jobs *manually* by manually deriving and setting the number of trainers. If this job fails due to OOM, the engineer must decide on a larger number of trainers for the job, and deploy it again. To reduce the number of failed trials, engineers often fine-tune the default number of trainers for a specific model based on their experience, and use the default value for future training jobs of that model.

The experience-based approach can hardly be accurate or keep up with the evolving models [92, 93]. In practice, engineers tend to over-provision GPUs to avoid job failures due to OOMs, just like any other hardware resources [9, 13, 40, 42, 77]. Our work is motivated by the need for an automatic GPU provisioning tool to accurately estimate the amount of GPU memory needed for a given job in order to improve GPU memory efficiency in production.

4 METHODOLOGY

Our study is based on data of DLRM offline training jobs from one of Meta's largest production clusters. The data range spans from February to December 2022; each part of the study is based on data spans of least 30 consecutive days.

Memory Efficiency Data. Our analysis on GPU memory efficiency (§5) is based on recorded resource monitoring data. The data records hourly-logged GPU memory utilization of all the deployed offline training jobs in the cluster for a limited time span. To analyze OOM events, we collect all the failed jobs due to OOM or sharding errors based on automated log analysis of error messages.

Monitoring. To conduct a more fine-grained analysis of GPU memory utilization (e.g., GPU memory utilization over time and at different stages, §6-8), we add GPU memory monitors [68] to the production training infrastructure to collect per-trainer memory utilization for all the deployed DLRM training jobs. The memory utilization of a GPU trainer is logged every 30 seconds (finest granularity without affecting production SLO) from model initialization to the end of the training loop. For example, to study GPU memory utilization before the training loop (§7), we signpost GPU memory utilization at important stages that occur before the training loop. We collect the per-trainer memory utilization right before and after the execution of each stage.

Profiling. We conduct experiments to profile memory usage over time for tensors computed in the forward and backward pass. We randomly sample 16 DLRM offline training jobs completed in November 2022 in the cluster—four jobs per each of the following DLRM: Model-A, Model-B, Model-C, and DHEN [97]. DHEN is included as a sub-model in Model-B (§3). For all 16 job samples, we profile memory behavior of their models for two training iterations. We empty the CUDA cache in between the two iterations [67]. We use the PyTorch profiler [73], which provides a decomposition of the allocated memory by tensor functionalities. We use profiling results from the second iteration to avoid temporal noise from bootstrapping the profiler and the training loop. For each sample, we control the profiling experiments on three aspects: (1) with or without materialized embedding tables, (2) varying three batch sizes, (3) single GPU versus single CPU. We have 12 ($2 \times 3 \times 2$) experiment runs per sample.

5 GPU MEMORY INEFFICIENCY

Recall from §3 that, without an effective GPU provisioning tool, engineers tend to over-provision GPUs to avoid job failures due to OOMs. While over-provisioning could benefit individual jobs, it leads to significant inefficiency of cluster-level GPU resources. In this section, we quantify the inefficiency in terms of GPU memory.

Finding 1. *50% of the DLRM offline training jobs utilize less than 50% of their provisioned GPU memory in production.*

The peak GPU memory utilization of a job is the peak GPU memory utilization averaged across its trainers, over the job's duration. The peak GPU memory utilization is below 50% for half of all the DLRM offline training jobs. Specifically, the 25th, 50th, 75th and 90th percentiles of peak utilization

are 25%, 43%, 71% and 95%. Under-utilization is more severe on jobs that are configured to use more trainers: less than 10% of the jobs that use more than 150 trainers had reached 50% peak GPU memory utilization. Engineers could intentionally over-provision GPUs for high throughput, e.g., to quickly test model accuracy. We opt for reducing cluster GPU memory inefficiency while meeting throughput requirement.

Finding 2. *31% of the failed DLRM offline training job attempts are due to preemption; 22% of the succeeded offline training jobs have been preempted at least once.*

Over-provisioning directly worsens GPU scarcity in production. Training jobs preempted due to GPU contention against a higher priority job are marked as failure, and will be requeued for resource. 31% of failed DLRM offline training job attempts are due to the job being preempted during execution for a higher-priority job. Further, 22% of the successfully completed DLRM offline training jobs have been preempted and retried at least once. The production cluster thus emits a behavior of *false idleness*: jobs often get preempted due to GPU contention, while majority of the jobs only utilize less than 50% of their provisioned GPU memory at best.

Finding 3. *8.67% of the failed DLRM offline training jobs are caused by GPU OOM in production.*

Although engineers often tend to over-provision GPUs, we still observe a fair amount of job failures due to under-provisioning. A job will fail due to GPU OOM when the memory usage on any one of the trainers exceeds its GPU memory capacity during training. We observe the percentage of GPU OOM in training job failures among Model-A, Model-B, Model-C, and all DLRMs are 15.13%, 16.39%, 6.94%, and 8.67%, respectively. 500,000+ GPU hours from the cluster were wasted within a 90-day range from July to September 2022. We also examine the occurrence of sharder failures due to GPU memory under-provisioning. A sharder fails if it cannot determine a sharding plan because the configured amount of trainers have insufficient memory to store embedding tables. Sharding failures are more lenient than OOM because they occur before training. Sharder failures constitute about 0.5% of the job failures, and wasted about 30,000 GPU hours from February to October 2022.

Implication. GPU memory under-utilization and OOM are prevalent in DNN training in open-source and production settings [19, 29, 30, 32]. We observe that large-scale distributed training simultaneously suffers from these consequences of GPU memory inefficiency. The demand for effective GPU memory provisioning is thus eminent for distributed training at scale, and has not been addressed in practice.

Distributed training jobs in production range from hundreds to thousands of GB in model size, and use tens to hundreds of GPUs in parallel. Accurately provisioning for such a diverse volume of jobs is challenging, because their GPU memory usage could be affected by various factors—parallelism paradigms, feature density, model architectures, etc. Depending on these factors, GPU memory usage differs significantly across jobs, and even across trainers of the same job. Moreover, a training job can carry arbitrary code and configuration changes, making manual solutions untenable. Most prior studies only provide theoretical reasoning on memory usage under limited scenarios (e.g., single-device). Unfortunately, the lack of understanding of actual GPU memory usage behavior of large-scale production workloads hinders solutions to GPU memory inefficiency in practice.

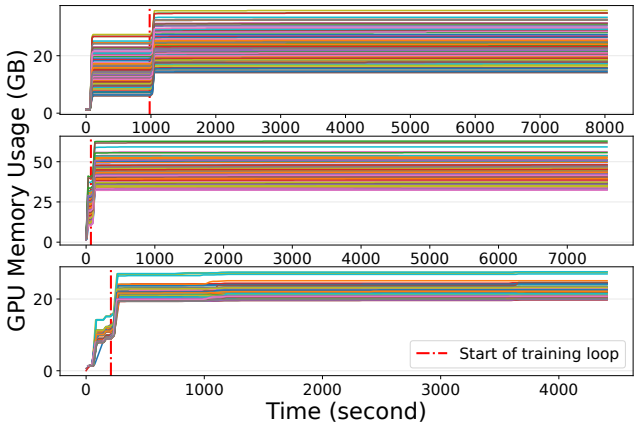

Figure 3: GPU memory utilization from model initialization till training ends for 3 randomly-sampled jobs. Top to bottom: Model-A, Model-B, and Model-C.

6 GPU MEMORY UTILIZATION

In this section, we conduct a fine-grained analysis of GPU memory utilization following §4. We calculate the GPU memory utilization of a trainer every 30 seconds as its *maximum* GPU memory usage within this 30-second window.

Finding 4. *GPU memory usage often varies significantly across trainers within a distributed training job in production.*

Figure 3 shows GPU memory utilization of three randomly sampled offline training jobs, one per our example DLRMs (§3). Each plot has 128 lines, each line shows the GPU memory utilization of a trainer. In each job sample, GPU memory usage varies significantly across trainers. This is common in jobs with model parallelism. On average, GPU memory usage of the most memory-consuming trainer is 47% higher than the average GPU memory usage across trainers in a job. This large variance is mainly caused by embedding tables varying across trainers (§7.3 provides detailed analysis).

Implication. GPU memory provisioning for large-scale distributed training should focus on the peak GPU memory

usage of the most memory-consuming trainer (especially for OOM prevention), instead of the average across trainers [19].

Our finding also shows that training jobs often do not use up the entire memory of every GPU. This presents potential for GPU memory sharing and job co-location for distributed training with model parallelism in practice, especially with the emergence of GPUs with larger HBM size [57]. Trainers with lower memory utilization can share their GPU memory within their GPU memory capacity. It also indicates opportunities for utilizing heterogeneous accelerators for distributed training under model parallelism. To achieve higher GPU memory utilization, GPU trainers with low utilization in a job can use GPU models that have a lower memory capacity during provisioning and scheduling stages. Existing work is limited to single-device or data parallelism-only scenarios, where GPU memory usage across trainers varies little.

Finding 5. *Most trainers stabilize at their peak memory usage early. In over 60% of the jobs: trainers reach their usage peak when the job is completed at most 40%, the most memory-consuming trainer reaches its usage peak when the job is completed at most 20%. See Figure 4.*

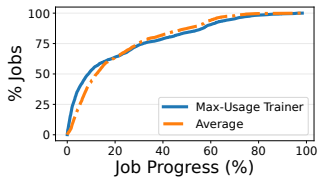

Figure 4: Job progress at 99% peak GPU mem usage.

In Figure 3, the GPU memory usage on each trainer of every job stabilizes quickly with minor fluctuation. Figure 4 shows how long a DLRM offline training job takes for its trainers to reach 99% of their peak GPU memory usage. "Max-Usage Trainer" shows how long the most memory-consuming trainer takes to reach 99% of its peak usage. "Average" shows how long *on average* all the trainers of a job take to reach 99% of their peak usage. Knowing the usage change over time for the most memory-consuming trainer in a distributed training job is important for GPU OOM prevention. In 60% or 80% of the jobs, the average trainers and the most memory-consuming trainer reach peak usage when the job is completed less than 20% or 40%, respectively. The minor bumps after usage stabilized are mainly caused by dynamic caching activities of the tensor allocator from PyTorch, which incrementally builds up a cache of CUDA memory and reassigns it to later allocations during training [59].

Implication. If peak GPU memory usage were highly unstable in large-scale distributed training, memory provisioning, sharing and balancing would have been difficult. We observe this is not the case. Building solutions to GPU memory inefficiency in large-scale distributed training is thus achievable.

Finding 6. *The memory usage incurred before and during the training loop emit distinct patterns. The memory usage incurred before training loop starts continues to constitute the peak overall memory usage during the training loop.*

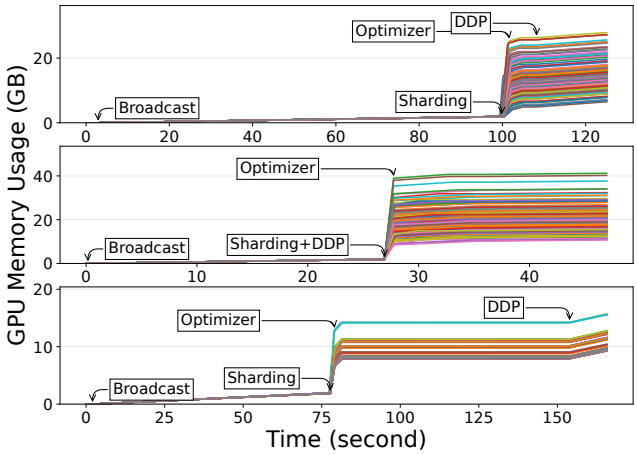

Figure 5: GPU memory usage over time prior to training loop for the same jobs in Figure 3.

We revisit Figure 3 to further analyze the peak overall GPU memory usage. Red vertical line separates the pre-training and training phases in each job (§2.2.1). These two phases have different memory usage patterns. Before training, there is a major memory usage increase, in which the memory usage on each trainer becomes significantly different. This increase is mainly due to model parallelism—embedding tables are partitioned and placed into each trainer's GPU memory. During training, there is another major memory usage increase on each trainer again. But the increased amount is approximately the same across trainers, and quickly stabilizes. This increase occurs as model starts running forward and backward pass with training samples on each trainer. Overall, memory usage incurred before training continues to constitute the total memory usage during training.

Implication. Our result motivates a divide-and-conquer approach to analyze the overall GPU memory usage. Distinct usage patterns before and during training prompt separate analyses of these two phases. And since they both contribute to the overall peak usage throughout training, their analysis results can be aggregated into analysis for the overall usage.

7 PRE-TRAINING GPU MEMORY USAGE

In this section, we study GPU memory usage incurred prior to the training loop (referred to as the *pre-training* phase).

Figure 5 shows GPU memory usage at important pre-training stages for the same jobs in Figure 3. The execution order and the memory usage at each stage could vary by models and jobs, because a training job can carry arbitrary changes from its engineer. From Figure 5, we observe that the memory usages for each stage together constitute the overall peak memory usage in the pre-training phase. Each stage

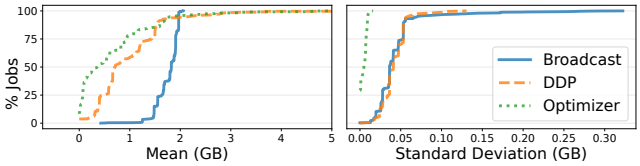

Figure 6: CDF of mean and stdev of GPU mem usage across trainers for "Broadcast", "Optimizer", "DDP".

also shows similar memory usage pattern across jobs. This observation motivates us to analyze each stage separately.

7.1 Broadcasting

Finding 7. *The memory usage for broadcasting in a distributed training job is within constant limit, while it can be positively affected by the number of trainers. See Figure 6 and 7.*

The "Broadcast" stage runs sharder and broadcasts metadata across trainers. Its memory usage is related to the size of the metadata, which includes sharding plan, data transformation information, and model configuration. Because the metadata is used in trainer communication, it constitutes the peak GPU memory usage of each trainer during training. We find metadata to be small so the memory usage for this stage is capped under a certain limit.

Figure 6 shows that the average GPU memory usage across trainers for "Broadcast" is almost the same in various jobs. Figure 7 shows GPU memory usage for "Broadcast" averaged across jobs under different numbers of trainers. "Broadcast" memory usage is overall higher in jobs that use more trainers, because metadata carries information of all trainers.

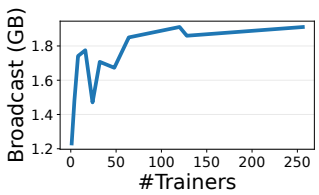

Figure 7: "Broadcast" GPU mem usage vs. #trainers.

Implication. We may account for "Broadcast" memory usage during provisioning by setting a constant upper-bound.

7.2 Optimizer and Data Parallelism

Finding 8. *The memory usages for constructing optimizer(s) and data parallelism both vary little across trainers. They are both linear to the model's dense parameters size. See Figure 8.*

The "Optimizer" stage constructs optimizer objects that will be used to update the model parameters during training [72]. The constructed objects hold the optimizer state (e.g., momentum [74], moments of the gradients [33]), and update the model parameters based on the computed gradients. Figure 8 compares the size of the dense parameters with the GPU memory usage of the "Optimizer" stage for 100 jobs. It shows that the size of the optimizer state is proportional to the size of the parameters it optimizes on, which is the size of the

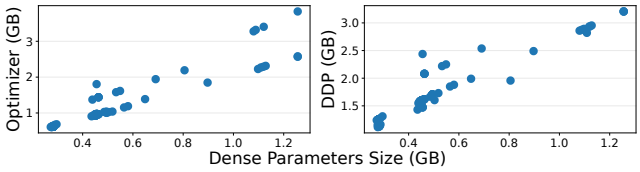

Figure 8: The size of dense parameters versus the max GPU memory usage across trainers for the "Optimizer" and "DDP" stages on 100 randomly-sampled jobs.

model's dense parameters in this case. Figure 6 further shows that the memory usage for "Optimizer" is the same across trainers within a job, as each trainer starts with optimizers that have the same initial state before training starts. Note that trainers here refer to GPUs, not the optimizer.

The "DDP" stage initializes data parallelism using DistributedDataParallel from PyTorch [36], which replicates the dense parameters across trainers, and creates necessary objects for gradient synchronization [64]. As shown in Figure 8, in each training job, the GPU memory usage for initializing DistributedDataParallel is approximately linear to the size of the dense parameters. We can also see from Figure 6 that the GPU memory usage of this stage has little variation across trainers in various jobs.

Implication. The memory usage for data parallelism and optimizer construction in distributed training jobs can be predicted linearly to the size of the model's dense parameters.

7.3 Model Parallelism

Finding 9. *Model parallelism incurs large difference across trainers on their memory usage for storing the model parallel component. In production, the standard deviation is up to 15GB, and the difference is up to 50GB, across trainers within the same distributed DLRM offline training job. See Figure 9.*

Recall from §6 that, GPU memory usage often varies significantly across trainers in a distributed training job. This is mostly due to the size of the model parallel component (e.g., embedding tables in DLRM) vary significantly across trainers under model parallelism. Figure 9 shows that the range and standard deviation of embedding table size across trainers in a job can be up to 50GB and 10GB, respectively. In 68% of the jobs, trainer storing the largest embedding table partition is the most-memory-consuming trainer.

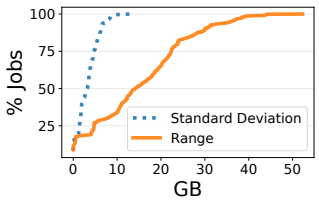

Figure 9: Range and stdev of Emb size across trainers.

In essence, model parallelism techniques often do not balance model size across trainers. In DLRM training specifically, embedding table lookup accounts for half of the total computation and communication cost [96]—they are accessed

across trainers in forward pass; their gradients are applied across trainers in backward pass (§2.2.1). Each embedding table largely differs in table-wise characteristics, e.g., dimension, lookup frequency (times a table is accessed), pooling factor (fetched rows per lookup), coverage (training samples accessing a table). Thus, how to shard terabytes of embedding tables across trainers significantly impacts training throughput. Currently, sharder's primary goal is maximizing training throughput, by partitioning the model such that *computation and communication costs of the partitions are balanced across trainers*. Finding an optimal sharding plan is NP-hard.

In existing sharders, model size (and its balance) is not a primary consideration with the goal of throughput maximization. To generate a sharding plan for DLRM, sharders use heuristic driven algorithms to partition and place embedding tables across trainers. Embedding tables can be partitioned at a mixture of column-, row- and table-wise granularities. Existing sharders [1, 41, 46, 79, 96] use various heuristics (e.g., table characteristics, hardware specifications) to define sophisticated cost functions; the optimization algorithms often have stochastic outputs (e.g., reinforcement learning). Under such design, lookup frequency and pooling factor are more effective as cost to be balanced towards generating a throughput-maximized sharding plan, as they directly represent the access frequency and access volume of an embedding table. But, lookup frequency and pooling factor often do not associate with table size. Figure 10 shows the normalized pooling factor, lookup frequency and size of 10% of the embedding tables in our studied jobs. A larger table does not mean it is used more frequently in training. Consequently, while pooling factor and lookup frequency are more likely to be balanced as part of the cost across trainers by sharders, embedding table size is not.

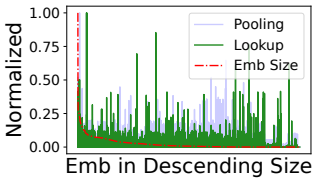

Figure 10: Emb table stats.

Note that some sharders use hierarchical sharding to balance model size across hosts to mitigate heavy inter-host communication. Hierarchical sharding assigns each host a similar load so that the inter-host communication is not bottlenecked by a few hosts that hold much more model parameters than the others. Figure 11 shows an offline training job sample with 128 trainers (16 hosts). Each dot shows the embedding table size on a trainer. We can see the embedding table size is approximately balanced across hosts.

Implication. Model parallelism techniques (i.e., sharders) often behave stochastically, and are too intricate to be modified for other purposes (e.g., provisioning). Instead, we can bypass handling the sharder's complexity while accounting for memory usage from model parallelism, by providing GPU memory provisioning results as input to the sharder to iteratively guide sharding and provisioning decisions. Since computation cost of a model parallel component may not correlate to its size, we believe future techniques on model parallelism should incorporate the goal of balancing the model parallel component size across trainers, which can reduce job-wise memory utilization imbalance in large-scale distributed training, and lead to higher training cluster efficiency.

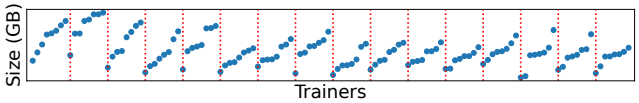

Figure 11: Model partition size per trainer of a random Model-A job with 128 trainers. Red lines separate hosts.

8 TRAINING-TIME GPU MEMORY USAGE

We now study the *new* GPU memory usage incurred during training. Note that pre-training usage (§7) still constitutes the overall GPU memory usage during training (§6). We count the peak training-time usage during job execution.

Finding 10. *Training-time usage varies little across trainers within a job. In 99% of the DLRM offline training jobs in production, the standard deviation is below 2GB. See Figure 12.*

The balance of training-time usage is attributed to the computation cost balance across trainers. Since the training phase only consists of running the distributed training loop, training-time usage is essentially the memory allocated for tensors generated while computing the forward and backward pass in each training iteration (§2.2.1). In DLRM training, the compute-intensive component (dense parameters) are replicated across trainers. So the training-time usage for computing forward/backward pass on the dense parameters are the same across trainers. Meanwhile, the capacity-demanding component (embedding tables) is partitioned across trainers by the sharder. As sharder optimizes computation-cost balance of embedding tables across trainers (§7.3), the training-time usage for embedding table tensor computations (i.e., lookup, pooling, weight update) during forward/backward pass is also opted for being balanced.

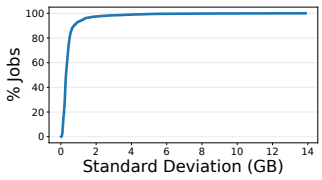

Figure 12: Training-time usage stdev across trainers.

Implication. Since trainers in a distributed training job often have the same training-time usage, we can apply the same analysis and techniques to estimate training-time usage across all trainers, rather than tackling it on individual trainers separately during GPU memory provisioning.

Balancing training-time usage across trainers offers convenience for tackling system problems in distributed training that often need memory estimation, e.g., GPU memory provisioning, scheduling, sharing, and balancing. As long as the model parallelism techniques can effectively balance the computation cost across trainers, training-time usage across trainers should be similar within a distributed training job.

8.1 Impact of Configurations

Finding 11. *Training-time usage variance across trainers in a job is primarily due to the variance of computational cost (e.g., lookup frequency and pooling factor in DLRM) across trainers.*

We found lookup frequency and pooling factor to be highly correlated—more frequently accessed embedding tables also have more rows fetched per access. Further, the lookup frequency and pooling factor of an embedding table is often significantly disproportional to its size. A small number of tables have higher access frequency and volume than the others in a DLRM, but are often not the largest in size (§7.3).

A trainer that stores embedding tables with higher lookup frequency and pooling factor should have a higher training-time usage, since more rows will be fetched/updated more frequently during training. Indeed, we observe a 0.30/0.13 Pearson coefficient between lookup frequency/pooling factor and training-time usage at trainer level. Since mixed use of model parallelisms (e.g., row-, column-, table-wise) introduces noise towards aggregating pooling factor and lookup frequency per trainer, we further validated the causality between trainer-wise lookup frequency (and pooling factor) and training-time usage using linear mixed effect model—a standard linear regression model used to determine the causal relationship between multiple variables.

Implication. Under model parallelism, imbalance of model parameter access frequency (e.g., lookup frequency) and access volume (e.g., pooling factor) causes the training-time usage to vary across trainers. Sharder often balances them (§7.3) and achieves a small training-time usage variance. Most often we can omit these factors while provisioning training-time usage. However, more fine-grained solutions should account for their influence to training-time usage, e.g., by iteratively optimizing provisioning decision with sharding decision until both reach convergence. Our results also motivate future work in feature engineering to design learning features with access frequency/volume proportional to their sizes for large-scale DNN, which helps to reduce sharding and load balancing complexity in distributed training.

Finding 12. *Training-time usage and its variance across trainers is linear to batch size. Meanwhile, part of training-time usage comes from communication overhead, which grows by the number of trainers within a constant limit.*

Table 3: Tensor categories.

Category	Description
Parameter	Model parameters used in forward pass, e.g., embedding tables and dense parameters (§2.1).
Gradient	Derivatives of prediction loss w.r.t. the model parameters.
Autograd	Tensors generated in backward pass that are not "Gradient" [62]. They are generally intermediate derivatives from the chain rule or implementation details of `Autograd`, e.g., accumulation buffers [61].
Activation	Tensors generated in forward pass, and is ultimately used to compute the gradient tensors.
Input	Tensors that contribute to forward pass and gradient computation, but do not depend on other tensors.
Temporary	Tensors created and destroyed in a single operator node.
Optimizer	Tensors held in the optimizer state which are used when an optimizer updates the model parameters.
Unknown	Tensors unfit for categories above, or memory allocation not matching a tensor's storage implementation [66].

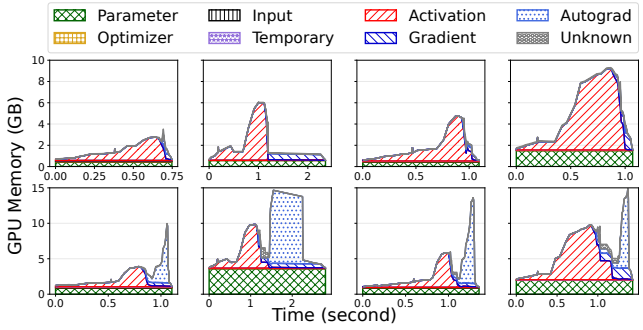

Figure 13: GPU memory usage decomposition by tensor functionality in one forward and backward pass. Each column from left to right shows one job sample for Model-A, Model-B, Model-C and DHEN. Jobs at the second row have embedding tables materialized as part of the model parameters, jobs at the first row do not.

We randomly selected one production offline training job per sample DLRM (§3), and deployed them with different batch sizes and number of trainers. Our experiments show the max, min, average, and variance of training-time usage across trainers are linear to batch size. The slope is different in each job due to model differences. Communication across trainers is frequently performed in each training iteration (§2.2.1), via PyTorch `AllReduce` [36] and NCCL [54]. Our experiments show communication overhead can grow from 1.5GB to 2GB per trainer when the same job uses more trainers.

Implication. We can provision training-time usage linear to batch size in distributed training jobs [19, 39, 78, 94]. The communication usage can be provisioned with a constant.

8.2 Decomposing Training-Time Usage

We further decompose training-time usage by profiling the memory usage of tensors computed in the forward and backward pass over time. We categorize the allocated memory

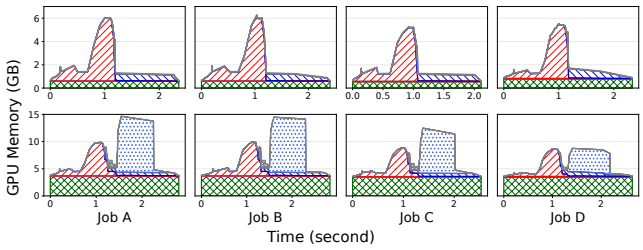

Figure 14: Four Model-B jobs under the same batch size. First/second row is without/with embedding tables.

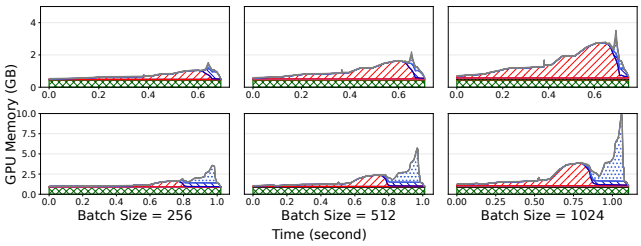

Figure 15: A Model-A job on batch size 256, 512 and 1024. First/second row is without/with embedding tables.

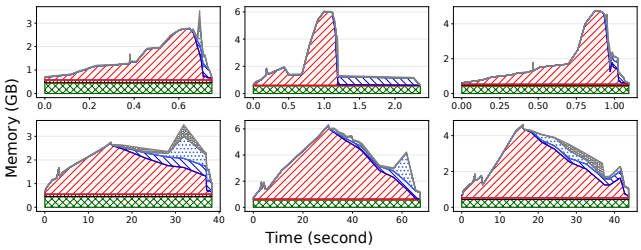

Figure 16: Three jobs without embedding tables running on GPU (first row), versus on CPU (second row). Left to right: Model-A, Model-B, Model-C.

by tensor functionality (see Table 3). Figure 13 presents the profiled GPU memory of four jobs, one per DLRM. All jobs use batch size 1024. Each color pattern indicates memory allocated for a tensor category.

Finding 13. *The peak training-time usage in each training iteration is dominated by the maximum of the peak forward pass memory usage and the peak backward pass memory usage.*

Figure 13 shows similar training-time usage pattern across different DLRMs. Activation, gradient and autograd tensors dominate the peak memory usage in training [38, 78, 90, 91]. In forward pass, activation tensors accumulate, whose memory usage surges quickly near the end of forward pass. This surge is mostly caused by large prediction head. In backward pass, autograd and gradient tensors w.r.t. the previously-computed activation tensors and parameters are being computed. Activation tensors are released after their corresponding gradients are computed. Gradient tensors are released after being applied to update parameters. Autograd tensors are not released until after the end gradients are derived [61]. Model parameters and input are kept until after training ends; memory usages of optimizer, temporary and unknown tensors are small. From Figure 13, the peak training-time usage equals to the maximum of the peak usage in forward pass and the peak usage in backward pass.

Training-time usage vastly differs across models, but varies little across jobs for the same model type. Figure 13 shows that each model has a different training-time usage peak and allocation trajectory, because model architecture governs the amount and size of tensors that will be traversed per training iteration. However, jobs for the same model type show similar pattern in Figure 14 (training jobs of the same model type often have small architectural changes).

In addition, we draw the following observations:

- In Figure 13 (first row), without materializing embedding tables, the memory bottleneck lies in caching and computation of the data parallel component. There is only one memory usage summit per training iteration, similar to profiling results on canonical DNNs [38, 91, 95]. In Figure 13 (second row), embedding tables are materialized in backward and forward pass. The memory bottleneck lies in caching and computation of the model parallel component. They largely increase the memory usage in backward pass, since many autograd tensors are held until gradients of the fetched table rows are derived. With embedding tables, there are two memory usage summits per iteration.
- The usage decomposition in Figure 15 shows that training-time usage is overall linear to batch size because both the peak usage for activation, and the peak usage for gradient and autograd, are linear to batch size.
- Figure 16 shows the peak training-time usage between GPU and CPU to be the same. But the usage accumulation trajectory differs, as activation and autograd tensors are held longer due to slower tensor computation on CPU.

Implication. Finding 13 offers a way to provision a model's training-time usage under given batch size. Because training-time usage pattern varies little across production jobs of the same model type, we can extend the provisioning system on a model basis rather than job basis. Since peak training-time usage is largely hardware-independent, we can omit hardware difference when provisioning training-time usage.

Our results also suggest that computation of the model parallel component incurs high memory usage in backward pass, resulting in multiple adjacent memory usage summits per training iteration. Existing memory sharing, or scheduling algorithms for DNN training jobs typically assume one isolated summit per iteration, with enough time in between to overlap multiple jobs on a single device [38, 91, 95]. Our results motivate future work on these directions to further account for this factor in model parallelism.

9 GPU MEMORY PROVISIONING

Recall from §3 that, manual GPU provisioning by engineers can hardly be accurate or keep up with evolving models, which leads to GPU memory inefficiency in large-scale distributed training (§5). In this section, we describe our experience of devising an effective GPU memory provisioning system for distributed training in production at Meta. Our system aims to achieve the following goals:

- *Accurate* provisioning of minimum GPU memory to meet throughput requirement, while avoiding OOM;
- *Automatic* provisioning without manual configuration or maintenance effort;
- *Generally applicable* to training jobs of different models;
- *Efficient* with little runtime overhead onto training.

9.1 Limitations of Existing Techniques

We started by exploring the viability of adopting existing work—a number of prior studies have presented approaches for estimating GPU memory usage for DNNs [2, 19, 39, 84, 94]. However, we find it challenging to adopt them in our production systems. From a high level, existing DNN memory usage estimation techniques can be categorized as follows:

9.1.1 **Modeling.** Several scheduling techniques estimate DNN memory usage with coarse-grained modeling derived from theoretical reasoning [2, 94]. They estimate a DNN's memory usage as the size sum of its parameters, input, activation and gradient tensors. More fine-grained estimation techniques [19, 84] define memory cost function (MCF) per operator. They define an operator's MCF as the sum of its weight, output and ephemeral tensor sizes (input, parameters, activation, gradient and temporary tensors in §8.2). These techniques statically traverse DNN graph, get memory cost at each operator by summing the sizes of its alive tensors according to its MCF (an alive tensor is one used by any current or descendent operator), then output the max memory cost across nodes as the estimated memory usage.

Limitations. Existing models miss important factors of the DNN memory usage. Coarse-grained models [2, 94] miss important components from the learning framework and training system, e.g., broadcast, optimizer (§7), and autograd (§8). Fine-grained models (i.e., MCFs) are static, and are derived by theoretically reasoning the relationship between the parameters and memory usage of each operator [19, 84]. However, operators in production are diverse and customized. Some operators have complex definitions, with stochastic behavior, e.g., the memory usage relation of embedding table operators varies by learning features [46]. Hence, hardcoding operator-wise models are brittle. Also, existing models cannot account for autograd tensors—their memory usage is dominant in backward pass and is unknown before execution (§8.2).

9.1.2 **Profiling.** Several scheduling techniques obtain memory usage of DNN jobs by performing fine-grained profiling on the jobs during training [18, 31, 35, 91]. They profile either the entire job, or smaller sample jobs of the actual job before it is scheduled, often at second or millisecond granularity.

Limitations. Profiling the training loop adds non-negligible overhead to training efficiency at production scale. A training iteration usually takes less than a second, job- or GPU-wise profiling at second granularity slows down the training loop for the vast volume of jobs that need profiling in the cluster. From our experience, profiling smaller sample jobs cannot provide accurate memory usage statistics for provisioning due to various shrinkage to the large-size model and training configurations of the actual job.

9.1.3 **Testing.** One technique [39] leverages test-case DNN generation and neural network formal specifications to estimate a model's memory usage. For a particular DNN (e.g., VGG [82]), it generates test-case DNNs by combinations of sample parameter inputs to each operator in the DNN. It then selects valid test-cases with formal specifications of the DNN architecture, obtain memory usage of each test-case via profiling or static analysis, and learns a polynomial regression on these test pairs to predict memory usage.

Limitations. With sophistical production model architectures [97], generating comparable test-cases is hard. Estimating memory usage of these test-cases also demand extra endeavors, i.e., profiling or static analysis [39]. Further, developing formal specifications of production models for test-case validation requires manual effort, and is difficult to keep up with the evolving models.

9.2 AMP: A Practical Provisioning System

We developed a practical GPU memory provisioning system called AMP for production DLRM training at Meta. AMP is an analytical approach built with open-source PyTorch.

9.2.1 **Design and Implementation.** AMP runs at model initialization. To provision GPU memory for a training job, AMP automatically estimates its model's peak memory usage given throughput requirement specified by engineers (batch size), and configures its number of trainers before training.

Design. Peak memory usage on a trainer is constituted by storing the model parallel component, and the other memory usage. Shown in Figure 17, AMP first estimates the *other memory usages* as the *reserved memory* of each trainer, and feeds it as input to the sharder. Sharder generates a sharding plan for model parallelism using the remaining memory (original capacity minus reserved memory) on each trainer. In this way, AMP bypasses interfering sharder's stochastic optimizations (§7.3). If sharder fails, AMP provisions more trainers from a small default value until sharder succeeds.

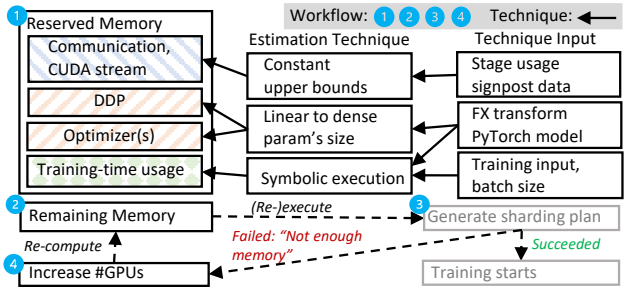

Figure 17: Provisioning system AMP first estimates the reserved memory, then iteratively provisions GPUs if remaining memory (memory capacity minus reserved memory) cannot fit the model parallel component.

AMP automatically estimates the reserved memory via divide-and-conquer ("Technique" in Figure 17). Peak memory usage accounted by the reserved memory is constituted by usages from pre-training stages, and training-time usage (§6). AMP estimates each of them separately:

- Peak memory usages for communication and CUDA stream are accounted by constant upper-bounds (§7.1, §8.1).
- Peak memory usages for "DDP" and "Optimizer" are estimated linear to the model's dense parameters size (§7.2).
- Training-time usage is estimated by symbolically executing the model under FX transformation [70] with training sample batches. AMP finds and compares the memory usage peaks in forward and backward pass via symbolic execution, and uses the higher peak as the estimate (§8).

Implementation. The upper-bounds for communication and CUDA stream usage are updated periodically by querying memory usage signposts at the start and end of the pre-training stages of recent historical job samples.

During model initialization of a job, we transform the model into a `torch.fx.GraphModule` instance, which is a graph intermediate representation of the model where each node represents a callsite to entities such as operator [70, 71]. We calculate the size of dense parameters on the transformed model to estimate memory usage for "DDP" and "Optimizer".

To estimate training-time usage, we symbolically execute the transformed model with raw training input batches. We use `ShapeProp` to execute the graph node-by-node with the given arguments, and get the output tensor's metadata (e.g., shape, data type, `require_grad`) of each node [65]. In the forward traversal, we compute the size-sum of output tensors that have their `require_grad` set to true as the *forward usage estimate*. This estimates the peak memory usage for activation. In the reversed traversal, we compute the maximum size-sum of output tensors used downstream at any given node in the graph as the *backward usage estimate*. This estimates the peak memory usage for gradient related tensors. The final estimated training-time usage is the maximum of

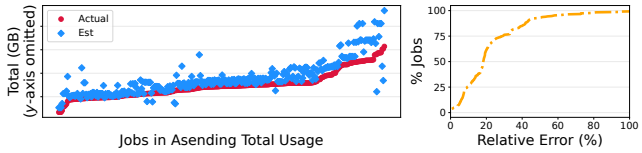

Figure 18: Actual versus estimated total GPU memory usage of 300 offline training jobs for top-five DLRMs.

these two entities (§8.2). Symbolically executing the model graph with original batch size (e.g., 1024, 2048) can cause OOM. So we compute estimates from 10 micro batch sizes (e.g., 13, 43), and use them to predict the final estimate (Finding 12). The estimation process takes a few seconds.

Summary. AMP provisions GPU memory by estimating and tuning the job configuration before training. It is thus easy to be incorporated into AutoML frameworks, e.g., for parameter auto-tuning, based on our experience in production.

AMP is independent of specific profilers, it only adds a one-time overhead to the job before training starts. AMP regularly queries memory usage signposts of pre-training stages from recent job samples (§3) to update upper-bound estimations with near-zero overhead. In comparison, profiling memory usage of a training job adds constant overhead to the job. AMP can account for memory usage from all important tensor categories missed by coarse-grained models (§8.2) with symbolic execution. AMP also does not need sophisticated models (i.e., operator-wise MCFs), because it symbolically executes each operator with training input and gathers their respective tensor sizes to compute usage peaks (§8.2). AMP requires no DNN generation and verification.

From our experience, AMP can be extended to other model types. If the model type has a new custom tensor type (which is uncommon), we need to add it to the tensor size-sum calculation when estimating training-time usage. Other usages differ little by model type. If a model type uses a new optimizer, we'll need to understand the optimizer as in §7.2.

AMP does not consider training-time usage variance across trainers. In practice, this variation is mostly very small (Figure 12), but it can be larger if computation cost is unbalanced (Finding 11). We can potentially address this issue by integrating sharding logic into provisioning. AMP also does not consider CUDA caching behavior which has minor impacts on total memory usage (§6).

9.2.2 **Evaluation.** We evaluate AMP in production for top-five DLRMs (§3). Figure 18 shows relative error ($\frac{100(Est-Actual)}{Actual}$) between the estimated and actual total GPU memory usage of the most memory-consuming trainer on 300 randomly sampled production offline training jobs in December 2022. The actual usage is obtained via monitoring (§4). For these jobs, the 75th percentile of symbolic execution cost is 2.2 seconds. The left plot compares actual and estimated usages,

where jobs are sorted in ascending actual usage. AMP is generally accurate, and over-estimates at the distribution tails. The right plot shows the estimate is below 25% over the actual max peak usage in 70% of the jobs. In 3% of the jobs, AMP under-estimated. In another 3%, the estimated usage exceeds the GPU memory capacity. In this case, user configurations are used as fallback. On the evaluated jobs, AMP can reduce user-configured number of GPUs by 45%, and save 1 million GBs of HBM.

10 DISCUSSION

In this section, we discuss the generalizability and limitations of our findings from §5-8, and threats to validity of this paper.

Our findings do not assume any characteristics of specific DLRMs. Embedding tables and other model components in a DLRM are essentially dense tensors, similar to what other major, large-scale DNN are composed of (§2.1). In DLRM, lookup frequency and pooling factor of the embedding tables represent the communication and computation costs of the dense tensors that are model-parallelized. Many of our findings should generalize to other DNNs beyond DLRMs.

Findings 1-3 study GPU memory inefficiency in production scale, which we believe is true for other DNNs [19, 29, 30, 32]. Findings 4-8 study memory usage governed by training components (training loop, broadcasting, optimization, data parallelism) whose implementations should generalize beyond specific model architectures [64, 69, 72]. Since our study is based on PyTorch models and tooling, these results could be specific to PyTorch and NCCL [54, 59, 62, 63, 73].

Findings 9-10 could be limited by the underlying model parallelisms. DLRM mostly uses intra-layer parallelisms [46], which shard each operator in the model into chunks [76, 81]. Orthogonal to intra-layer, inter-layer parallelisms (e.g., pipeline parallelism) shard the model into groups of operators [27, 48]. Some techniques use both [50, 99]. Under inter-layer parallelisms, memory usage could still differ across GPUs due to sharding, but the training-time usage may vary across GPUs, because each GPU holds a different group of operators and the computation cost may vary across GPUs.

Findings 11-12 study the effect of computation cost balance, batch size, and number of trainers in large-scale distributed training. We believe their relations to GPU memory usage are generalizable. Finding 13 generalizes beyond DLRM from our experience, though concrete memory usage shape would differ in different DNN architectures.

11 OTHER RELATED WORK

Prior studies on DNN training cluster focus on workload characteristics, cluster throughput and training failures [1, 6, 25, 30]. Our work complements them with in-depth characterization of DNN training memory usage (§5-6).

DNN memory management techniques aim to reduce model memory footprint (the amount of memory a model needs) to improve training throughput [16, 37, 76, 78, 90]. They offer insights on DNN memory usage with theoretical reasoning and static analysis. We aim to reduce the amount of memory a job requests to improve GPU memory efficiency, with a more fine-grained analysis at production scale.

Memory sharing for concurrent training [38, 43, 56, 91, 95] leverages the cyclic memory usage pattern of DNN to enable a GPU's memory shared temporally (time slicing) or spatially by multiple training jobs on single-device or data-parallel scenarios. We study memory usage of large-scale production DNN training jobs under mixed parallelisms. Our results show distinct characteristics of model-parallel jobs, and potentials for future work in memory sharing (§6, §7.3).

DNN job scheduling and balancing techniques assume requested amount of GPU memory of a training job to be fixed or predefined, and focus on maximizing training cluster throughput [2, 3, 6, 17, 18, 18, 20, 28, 31, 35, 45, 49, 60, 91, 94]. Our work focuses on proactively reducing the requested amount of GPU memory of a job. We analyze GPU memory efficiency of production jobs under mixed parallelisms to shed light on future work along distributed training at scale.

12 CONCLUSION

As the size of deep learning models grows at terabyte scale, large-scale distributed training is becoming a norm. This paper presents a systematic analysis of GPU memory behavior of large-scale distributed training jobs in production at Meta. We measure GPU memory inefficiency, characterize GPU memory utilization, and provide fine-grained analysis on GPU memory usage of production jobs. Our study revealed over a dozen findings with concrete implications. We further build on the analysis to develop a practical GPU provisioning system to improve GPU memory efficiency in production. We believe future research can leverage our study to improve GPU memory efficiency in distributed training.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Venkata Vamsikrishna Meduri, for their valuable comments. We thank Xiaodong Wang, Menglu Yu, Taylor Robie, Srinath Mandalapu, Mehmet Yunt, and other Meta colleagues for their helpful discussions and infrastructure support. We thank Gangmuk Lim, Chirag Shetty, and Darko Marinov for discussions on early drafts. Runxiang Cheng is supported in part by NSF grant CCF-1763788. Tianyin Xu is supported in part by NSF grants CNS-2145295 and CNS-1956007.

REFERENCES

[1] Acun, B., Murphy, M., Wang, X., Nie, J., Wu, C.-J., and Hazelwood,

K. Understanding training efficiency of deep learning recommendation models at scale. In *HPCA* (2021).

[2] Albahar, H., Dongare, S., Du, Y., Zhao, N., Paul, A. K., and Butt, A. R. Schedtune: A heterogeneity-aware gpu scheduler for deep learning. In *CCGrid* (2022).

[3] Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys* (2022).

[4] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv:2108.07258* (2021).

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *NeurIPS* (2020).

[6] Chen, Z., Quan, W., Wen, M., Fang, J., Yu, J., Zhang, C., and Luo, L. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *TPDS* (2019).

[7] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al. Wide & deep learning for recommender systems. In *DLRS* (2016).

[8] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv:2204.02311* (2022).

[9] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP* (2017).

[10] Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In *RecSys* (2016).

[11] Davis, W., and Lawler, R. Nvidia became a $1 trillion company thanks to the AI boom. https://www.theverge.com/2023/5/30/23742123/nvidia-stock-ai-gpu-1-trillion-market-cap-price-value, 2023.

[12] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *NeurIPS* (2012).

[13] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* (2014).

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

[15] Elkahky, A. M., Song, Y., and He, X. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *WWW* (2015).

[16] Fang, J., Zhu, Z., Li, S., Su, H., Yu, Y., Zhou, J., and You, Y. Parallel training of pre-trained models via chunk-based dynamic memory management. *TPDS* (2022).

[17] Gao, W., Hu, Q., Ye, Z., Sun, P., Wang, X., Luo, Y., Zhang, T., and Wen, Y. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv:2205.11913* (2022).

[18] Gao, W., Ye, Z., Sun, P., Wen, Y., and Zhang, T. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *SoCC* (2021).

[19] Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating gpu memory consumption of deep learning models. In *ESEC/FSE* (2020).

[20] Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H. H., and Guo, C. Tiresias: A gpu cluster manager for distributed deep learning. In *NSDI* (2019).

[21] Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., et al. The architectural implications of facebook's dnn-based personalized recommendation. In *HPCA* (2020).

[22] Habib, R. OpenAI's plans according to Sam Altman. https://website-754fwhahs-humanloopml.vercel.app/blog/open_ai_talk?utm_source=bensbites&utm_medium=newsletter&utm_campaign=openai-s-roadmap/, 2023.

[23] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR* (2016).

[24] Hsia, S., Gupta, U., Wilkening, M., Wu, C.-J., Wei, G.-Y., and Brooks, D. Cross-stack workload characterization of deep recommendation systems. In *IISWC* (2020).

[25] Hu, Q., Sun, P., Yan, S., Wen, Y., and Zhang, T. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *SC* (2021).

[26] Huang, S., Dong, L., Wang, W., Hao, Y., Singhal, S., Ma, S., Lv, T., Cui, L., Mohammed, O. K., Liu, Q., et al. Language is not all you need: Aligning perception with language models. *arXiv:2302.14045* (2023).

[27] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS* (2019).

[28] Hwang, C., Kim, T., Kim, S., Shin, J., and Park, K. Elastic resource sharing for distributed deep learning. In *NSDI* (2021).

[29] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. A comprehensive study on deep learning bug characteristics. In *ESEC/FSE* (2019).

[30] Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *ATC* (2019).

[31] Jia, X., Jiang, L., Wang, A., Xiao, W., Shi, Z., Zhang, J., Li, X., Chen, L., Li, Y., Zheng, Z., et al. Whale: Efficient giant model training over heterogeneous gpus. In *ATC* (2022).

[32] Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., and Guo, C. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI* (2020).

[33] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv:1412.6980* (2014).

[34] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997* (2014).

[35] Le, T. N., Sun, X., Chowdhury, M., and Liu, Z. Allox: compute allocation in hybrid clusters. In *EuroSys* (2020).

[36] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv:2006.15704* (2020).

[37] Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *arXiv:2202.01306* (2022).

[38] Lim, G., Ahn, J., Xiao, W., Kwon, Y., and Jeon, M. Zico: Efficient gpu memory sharing for concurrent dnn training. In *ATC* (2021).

[39] Liu, H., Liu, S., Wen, C., and Wong, W. E. Tbem: Testing-based gpu-memory consumption estimation for deep learning. *IEEE Access* (2022).

[40] Liu, Q., and Yu, Z. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *SoCC* (2018).

[41] Lui, M., Yetim, Y., Ozkan, O., Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. Understanding capacity-driven scale-out neural recommendation inference. In *ISPASS* (2021).

[42] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., and Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC* (2021).

[43] Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient gpu cluster scheduling. In *NSDI* (2020).

[44] Mark Harris. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.
[45] Mohan, J., Phanishayee, A., Kulkarni, J., and Chidambaram, V. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *OSDI* (2022).
[46] Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA* (2022).
[47] Mujtaba, H. NVIDIA AI GPU Demand Blows Up, Chip Prices Increase By 40% & Stock Shortages Expected Till December. https://wccftech.com/nvidia-ai-gpu-demand-blows-up-chip-prices-increase-40-percent-stock-shortages-till-december/, 2023.
[48] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP* (2019).
[49] Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI* (2020).
[50] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SC* (2021).
[51] Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv:2003.09518* (2020).
[52] Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv:1906.00091* (2019).
[53] NVIDIA. NVIDIA V100 Tensor Core GPU Datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf/, 2018.
[54] NVIDIA. NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl, 2019.
[55] NVIDIA. NVIDIA A100 Tensor Core GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf/, 2020.
[56] NVIDIA. CUDA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2022.
[57] NVIDIA. NVIDIA GH200 Grace Hopper Superchip. https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip, 2023.
[58] OpenAI. GPT-4 Technical Report. https://cdn.openai.com/papers/gpt-4.pdf, 2023.
[59] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS* (2019).
[60] Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys* (2018).
[61] Autograd mechanics. https://pytorch.org/docs/stable/notes/autograd.html, 2023.
[62] Automatic differentiation package. https://pytorch.org/docs/stable/autograd.html, 2023.
[63] CUDA semantics. https://pytorch.org/docs/stable/notes/cuda.html, 2023.
[64] DistributedDataParallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html, 2023.
[65] Shape propagation. https://github.com/pytorch/pytorch/blob/master/torch/fx/passes/shape_prop.py, 2023.
[66] StorageImpl. https://github.com/pytorch/pytorch/blob/master/c10/core/StorageImpl.h, 2023.
[67] torch.cuda.empty_cache. https://pytorch.org/docs/stable/generated/torch.cuda.empty_cache.html, 2023.
[68] torch.cuda.list_gpu_processes. https://pytorch.org/docs/stable/generated/torch.cuda.list_gpu_processes.html, 2023.
[69] torch.distributed.broadcast_object_list. https://pytorch.org/docs/stable/distributed.html#torch.distributed.broadcast_object_list, 2023.
[70] torch.fx. https://pytorch.org/docs/stable/fx.html, 2023.
[71] torch.fx.Node. https://pytorch.org/docs/stable/fx.html#torch.fx.Node, 2023.
[72] torch.optim.Optimizer. https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer, 2023.
[73] torch.profiler. https://pytorch.org/docs/stable/profiler.html, 2023.
[74] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks* (1999).
[75] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR* (2020).
[76] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC* (2020).
[77] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC* (2012).
[78] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO* (2016).
[79] Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., and Wu, C.-J. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. In *ASPLOS* (2022).
[80] Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *NeurIPS* (2018).
[81] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv:1909.08053* (2019).
[82] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2014).
[83] Steck, H., Baltrunas, L., Elahi, E., Liang, D., Raimond, Y., and Basilico, J. Deep learning for recommender systems: A netflix case study. *AI Magazine* (2021).
[84] Sun, Q., Liu, Y., Yang, H., Zhang, R., Dun, M., Li, M., Liu, X., Xiao, W., Li, Y., Luan, Z., et al. Cognn: efficient scheduling for concurrent gnn training on gpus. In *SC* (2022).
[85] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *CVPR* (2016).
[86] DistributedModelParallel. https://pytorch.org/torchrec/torchrec.distributed.html#torchrec.distributed.model_parallel.DistributedModelParallel, 2023.
[87] EmbeddingShardingPlanner. https://pytorch.org/torchrec/torchrec.distributed.planner.html#torchrec.distributed.planner.planners.EmbeddingShardingPlanner, 2023.
[88] TorchRec. https://github.com/pytorch/torchrec, 2023.
[89] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *NeurIPS* (2017).
[90] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. Superneurons: Dynamic gpu memory management for training deep neural networks. In *PPoPP* (2018).
[91] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva:

Introspective cluster scheduling for deep learning. In *OSDI* (2018).

[92] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do not blame users for misconfigurations. In *SOSP* (2013).

[93] Xu, T., and Zhou, Y. Systems approaches to tackling configuration errors: A survey. *CSUR* (2015).

[94] Yeung, G., Borowiec, D., Yang, R., Friday, A., Harper, R., and Garraghan, P. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *TPDS* (2021).

[95] Yu, P., and Chowdhury, M. Fine-grained gpu sharing primitives for deep learning applications. *NeurIPS* (2020).

[96] Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhushanam, B., Tian, Y., Kejariwal, A., and Hu, X. Dreamshard: Generalizable embedding table placement for recommender systems. *arXiv:2210.02023* (2022).

[97] Zhang, B., Luo, L., Liu, X., Li, J., Chen, Z., Zhang, W., Wei, X., Hao, Y., Tsang, M., Wang, W., et al. Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction. *arXiv:2203.11014* (2022).

[98] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *MLSys* (2020).

[99] Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *OSDI* (2022).