

# DISTRIBUTED HIERARCHICAL GPU PARAMETER SERVER FOR MASSIVE SCALE DEEP LEARNING ADS SYSTEMS

Wei jie Zhao<sup>1</sup>, Deping Xie<sup>2</sup>, Ronglai Jia<sup>2</sup>, Yulei Qian<sup>2</sup>, Ruiquan Ding<sup>3</sup>, Mingming Sun<sup>1</sup>, Ping Li<sup>1</sup>

<sup>1</sup> Cognitive Computing Lab, Baidu Research

<sup>2</sup> Baidu Search Ads (Phoenix Nest), Baidu Inc.

<sup>3</sup> Sys. & Basic Infra., Baidu Inc.

{weijiezhao, xiedeping01, jiaronglai, qianyulei, dingruiquan, sunmingming01, liping11}@baidu.com

## ABSTRACT

Neural networks of ads systems usually take input from multiple resources, e.g., query-ad relevance, ad features and user portraits. These inputs are encoded into one-hot or multi-hot binary features, with typically only a tiny fraction of nonzero feature values per example. Deep learning models in online advertising industries can have terabyte-scale parameters that do not fit in the GPU memory nor the CPU main memory on a computing node. For example, a sponsored online advertising system can contain more than  $10^{11}$  sparse features, making the neural network a massive model with around 10 TB parameters. In this paper, we introduce a distributed GPU hierarchical parameter server for massive scale deep learning ads systems. We propose a hierarchical workflow that utilizes GPU High-Bandwidth Memory, CPU main memory and SSD as 3-layer hierarchical storage. All the neural network training computations are contained in GPUs. Extensive experiments on real-world data confirm the effectiveness and the scalability of the proposed system. A 4-node hierarchical GPU parameter server can train a model more than 2X faster than a 150-node in-memory distributed parameter server in an MPI cluster. In addition, the price-performance ratio of our proposed system is 4-9 times better than an MPI-cluster solution.

## 1 INTRODUCTION

Baidu Search Ads (a.k.a. “Phoenix Nest”) has been successfully using ultra-high dimensional input data and ultra-large-scale deep neural networks for training CTR (Click-Through Rate) models since 2013 (Fan et al., 2019). Sponsored online advertising produces many billions of dollar revenues for online ad publishers such as Baidu, Bing, and Google. The task of CTR prediction (Broder, 2002; Fain & Pedersen, 2006; Fan et al., 2019; Zhao et al., 2019) plays a key role to determine the best ad spaces allocation because it directly influences user experience and ads profitability. CTR prediction takes input from multiple resources—e.g., query-ad relevance, ad features, and user portraits—then estimates the probability that a user clicks on a given ad. These inputs are typically one-hot/multi-hot binary features with only a tiny fraction of nonzero feature values per example. The high-dimensional sparse input data first go through an embedding layer to get a low-dimensional embedding; then the embedded results are connected by fully-connected layers. A real-world sponsored online advertising system can contain more than  $10^{11}$  sparse features, making the neural network a massive model with around 10 TB parameters. Figure 1 provides an illustration of CTR prediction network.

*Proceedings of the 3<sup>rd</sup> MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).*

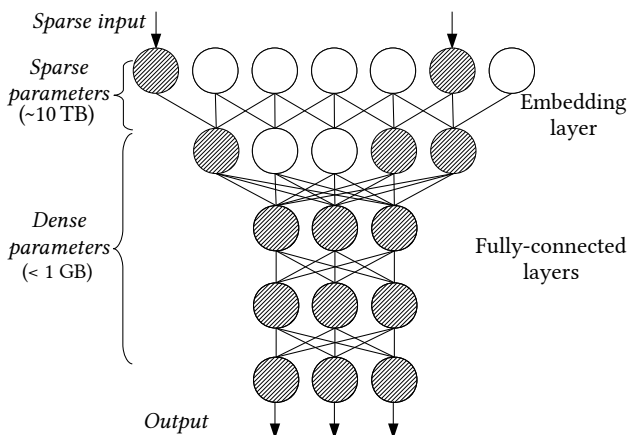


Figure 1. A visual illustration of CTR prediction network. The input data are sparse—only a tiny proportion of features are non-zero. The parameters of hatched neurons are referenced by the input.

### 1.1 CTR Prediction Based on MPI-Cluster Solution

Since 2013, the standard practice in Baidu Search Ads has been using an MPI-based solution for training massive-scale deep learning CTR prediction. Before that, Baidu had adopted a distributed logistic regression (LR) CTR model and distributed parameter server around 2010. The MPI solution partitions the model parameters across computing nodes (e.g., 150 nodes). Each node obtains a batch of

training data streamed from a distributed file system. The node pulls the required parameters from other nodes and computes the gradients. The gradients are pushed to the nodes that maintain the corresponding parameters through MPI communications. Baidu’s MPI solution and distributed parameter server was also summarized in (Li et al., 2014) (while the first author in that paper was with Baidu Inc.).

## 1.2 Drawbacks of MPI-Cluster Solution

CTR model researchers and practitioners train the massive-scale model rapidly to test and verify their new ideas. However, it is impractical to reserve a large-scale computing cluster for everyone—the hardware maintenance and the energy cost of a large-scale computing cluster is pricey. Besides, further scaling up the training speed is challenging—in this large-scale distributed setting, communication and synchronization cost between computing nodes limits the training performance. We seek to explore other directions to optimize the training system for massive-scale ads models.

## 1.3 GPU Parameter Server for CTR Prediction

Recently, GPUs attract broad attention as a cost-efficient machine learning accelerator. However, it is not straightforward to directly employ GPUs on the CTR prediction problem due to the limited GPU memory and excessive CPU-GPU data communications. This amount of parameters exceeds the amount of overall volume of GPU High-Bandwidth Memory (HBM) of any single computing node, and any practical computing cluster. GPUs are considered inefficient to train the model that does not fit in the GPU memory (Chilimbi et al., 2014). Prior efforts (Cui et al., 2016; Zhao et al., 2019) cache the parameters in the CPU main memory and still require time-consuming data transferring between CPU and GPU to update parameters.

We notice that in the massive-scale ads system, although the entire model is huge, the number of non-zero features per example is small. For example, in Figure 1, the referenced parameters for the given input are hatched. The sparse input has 2 non-zeros: the 1<sup>st</sup> and the 6<sup>th</sup> features. For the parameters in the embedding layer, only a subset of them is used and will be updated for this sparse input (we call them “sparse parameters” for convenience). Following that, the dense parameters in the fully-connected layers are all referenced. The CTR prediction neural network usually has at most a few million dense parameters—the memory footprint of dense parameters is small (around a few hundred megabytes). The working parameters—the parameters that are referenced in the current batch input—can fit in the GPU memory. It allows us to distribute the working parameters in the GPU memory and accelerate the training with GPUs.

In this paper, we propose a hierarchical parameter server that builds a distributed hash table across multiple GPUs

and performs direct inter-GPU communications to eliminate the CPU-GPU data transferring overhead. Moreover, the hierarchical design on top of SSDs with an in-memory cache also enables us to train a large-scale out-of-main-memory deep neural network on distributed GPU environments.

**Challenges & approaches.** There are three major challenges in the design of the hierarchical parameter server. 1) The first challenge is to construct an efficient distributed GPU hash table to store the working parameters in multiple GPUs in multiple computing nodes. We design intra-node GPU communications and collective communication strategies among multiple nodes to synchronize the parameters across all GPUs. Remote direct memory access protocols are employed to enable direct peer-to-peer GPU communications without involving CPUs. 2) The second challenge is to devise a data transferring and caching mechanism to keep working parameters in memory. Data transferring and SSD I/Os bandwidth are relatively slow compared with the deep neural network training on GPUs. Without a carefully designed architecture, excessive network and disk I/Os can be a dominant performance slowdown. We propose a 4-stage pipeline that overlaps inter-node network communications, SSD I/Os, CPU/GPU data transferring and GPU computations. All stages are optimized to match the inevitable GPU computations—the pipeline hides the I/O latency. 3) The third challenge is to effectively organize the materialized parameters on SSDs. The I/O granularity of SSDs is a block, while the training parameters we load is in key-value granularity. This mismatch incurs I/O amplification (Lu et al., 2017)—unnecessary data are read from SSDs to load required parameters. In contrast, reading and writing contiguous addresses on SSDs are faster than random accesses—larger block size yields better disk I/O bandwidth. We present a file-level parameter management strategy to write updates in batches as new files. A file compaction thread runs in the background to regularly merge files that contain a large proportion of stale parameters into new files.

**Contributions.** The technical contributions we make in this paper can be summarized as follows:

- We present a hashing method (OP+OSRP) to reduce CTR models. We observe that combining OP+OSRP with DNN-based models can replace the original LR-based models. However, for DNN-based CTR models, the hashing method will hurt the prediction accuracy to the extent that the revenue would be noticeably affected (Section 2).
- We introduce the architecture of a distributed hierarchical GPU parameter server (Section 3). The hierarchical design employs the main memory and SSDs to store the out-of-GPU-memory and out-of-main-memory parameters when training massive-scale neural networks. As far as we know, it is the first distributed GPU parameter server for the terabyte-scale sparse input deep learning.

- We propose a novel HBM parameter server that keeps the working parameters in distributed GPU HBMs. The HBM parameter server enables GPU worker threads to efficiently fetch/update neural network parameters and to directly synchronize parameters across multiple GPU cards from multiple computing nodes (Section 4).
- We show a CPU main memory parameter server that prefetches and buffers the parameters of the future training batches. The memory parameter server hides the data loading and parameter transfer latency (Section 5).
- We present an SSD parameter server that materializes the out-of-main-memory parameters in files. Parameter updates are written in batches onto SSDs as new files to fully utilize the I/O bandwidth. Regular compaction operations are performed to bound the disk usage (Section 6).
- We perform an extensive set of experiments on 5 real CTR prediction models/datasets and compare the results with an MPI cluster training framework in the production environment (Section 7). The 4-node distributed hierarchical GPU parameter server is 2.4-4.8X faster than the MPI solution. The price-performance ratio of our proposed system is 4-9 times better than the MPI solution.

## 2 PRIOR EFFORT: HASHING METHODS FOR REDUCING CTR MODELS

It is natural to ask why one could not simply use a good hashing method to reduce the model size. In this section, we report a brief summary of our prior efforts back in 2015 for developing hashing algorithms to reduce the size of CTR models, with the hope that hashing would not hurt accuracy.

Hashing algorithms are popular these days (Weinberger et al., 2009; Li et al., 2011). We had spent serious efforts to develop effective hashing methods suitable for CTR models. The results are both encouraging and discouraging. Here we would like to make a comment that there is a difference between academia research and industrial practice. In academia research, we often take it as a good result if, for example, we are able to achieve a 10-fold model reduction by losing only 1% of accuracy. For commercial search, however, we often cannot afford to lose an accuracy of 0.1% since that would result in a noticeable decrease in revenue.

**One permutation + one sign random projection.** In the course of this research, we had tried many versions of hashing. We eventually recommend the method called “one permutation + one sign random projection (OP+OSRP)”. As the name suggests, our idea was inspired by several well-known hashing methods, e.g., (Charikar et al., 2004; Cormode & Muthukrishnan, 2005; Weinberger et al., 2009; Li et al., 2011; 2012; Shrivastava & Li, 2015; Li et al., 2019).

We assume the dimensionality of the binary training data is  $p$ . OP+OSRP consists of the following key steps:

1. Permute the  $p$  columns (only once). This can be very efficiently done via standard 2U or 4U hashing.
2. Break the permuted  $p$  columns uniformly into  $k$  bins.
3. Apply independent random projection in each bin, for example, in the first bin, we let  $z = \sum_{i=1}^{p/k} x_i r_i$ , where we sample  $r_i \in \{-1, +1\}$  w. p.  $1/2$ .
4. Store the sign of  $z$  and expand it into a vector of length 2:  $[0 \ 1]$  if  $z > 0$ ,  $[1 \ 0]$  if  $z < 0$ ,  $[0 \ 0]$  if  $z = 0$ . The hashed data will be binary in  $2k$  dimensions. This step is different from prior research. This way, we still obtain binary features so that we do not need to modify the training algorithm (which is coded efficiently for binary data).

OP+OSRP is very efficient (essentially by touching each nonzero entry once) and can naturally handle sparse data (i.e., many bins will be empty since  $k$  cannot be small).

**Experimental results.** This set of experiments was conducted in 2015 using 3 months of sponsored ads click data from web search and 3 months of data from image search. As one would expect, web search brings in majority of the revenue. Image search is fun and important but its revenue is only a tiny fraction of the revenue from web search. Understandably, lots of computing/storage resources have been allocated for building CTR models for web search.

Table 1. OP+OSRP for Image Search Sponsored Ads Data

	# Nonzero Weights	Test AUC
Baseline LR	31,949,213,205	0.7112
Baseline DNN		0.7470
Hash+DNN ( $k = 2^{34}$ )	6,439,972,994	0.7407
Hash+DNN ( $k = 2^{23}$ )	3,903,844,565	0.7388
Hash+DNN ( $k = 2^{22}$ )	2,275,442,496	0.7370
Hash+DNN ( $k = 2^{31}$ )	1,284,025,453	0.7339
Hash+DNN ( $k = 2^{30}$ )	707,983,366	0.7310
Hash+DNN ( $k = 2^{29}$ )	383,499,175	0.7278
Hash+DNN ( $k = 2^{28}$ )	203,864,439	0.7245
Hash+DNN ( $k = 2^{27}$ )	106,824,123	0.7208
Hash+DNN ( $k = 2^{26}$ )	55,363,771	0.7175
Hash+DNN ( $k = 2^{25}$ )	28,479,330	0.7132
Hash+DNN ( $k = 2^{24}$ )	14,617,324	0.7113

Table 1 summarizes the experimental result for applying OP+OSRP on image search ads data. There are several important observations from the results:

- Compared to the baseline LR model, the DNN model very substantially improves the AUC. This is basically the justification of adopting DNN models for CTR prediction.
- Hashing reduces the accuracy. Even with  $k = 2^{34}$ , the test AUC is dropped by 0.7%.
- Hash+DNN is a good combination for replacing LR. Compared to the original baseline LR model, we can reduce the number of nonzero weights from 31B to merely 14.6M without affecting the accuracy.

Table 2 summarizes the experiments on web search ads data. The trend is essentially similar to Table 1. The main difference is that we cannot really propose to use Hash+DNN for web search ads CTR models, because that would reduce the accuracy of current DNN-based models and consequently would affect the revenue for the company.

Table 2. OP+OSRP for Web Search Sponsored Ads Data

	# Nonzero Weights	Test AUC
Baseline LR	199,359,034,971	0.7458
Baseline DNN		0.7670
Hash+DNN ( $k = 2^{32}$ )	3,005,012,154	0.7556
Hash+DNN ( $k = 2^{31}$ )	1,599,247,184	0.7547
Hash+DNN ( $k = 2^{30}$ )	838,120,432	0.7538
Hash+DNN ( $k = 2^{29}$ )	433,267,303	0.7528
Hash+DNN ( $k = 2^{28}$ )	222,780,993	0.7515
Hash+DNN ( $k = 2^{27}$ )	114,222,607	0.7501
Hash+DNN ( $k = 2^{26}$ )	58,517,936	0.7487
Hash+DNN ( $k = 2^{24}$ )	15,410,799	0.7453
Hash+DNN ( $k = 2^{22}$ )	4,125,016	0.7408

**Summary.** This section summarizes our effort on developing effective hashing methods for ads CTR models. The work was done in 2015 and we had never attempted to publish the paper. The proposed algorithm, OP+OSRP, actually still has some novelty to date, although it obviously combines several previously known ideas. The experiments are exciting in a way because it shows that one can use a single machine to store the DNN model and can still achieve a noticeable increase in AUC compared to the original (large) LR model. However, for the main ads CTR model used in web search which brings in the majority of the revenue, we observe that the test accuracy is always dropped as soon as we try to hash the input data. This is not acceptable in the current business model because even a 0.1% decrease in AUC would result in a noticeable decrease in revenue.

Therefore, this report helps explain why we introduce the distributed hierarchical GPU parameter server in this paper to train the massive scale CTR models, in a lossless fashion.

### 3 DISTRIBUTED HIERARCHICAL PARAMETER SERVER OVERVIEW

In this section, we present the distributed hierarchical parameter server overview and describe its main modules from a high-level view. Figure 2 illustrates the proposed hierarchical parameter server architecture. It contains three major components: HBM-PS, MEM-PS and SSD-PS.

**Workflow.** Algorithm 1 depicts the distributed hierarchical parameter server training workflow. The training data batches are streamed into the main memory through a network file system, e.g., HDFS (line 2). Our distributed training framework falls in the data-parallel paradigm (Li et al., 2014; Cui et al., 2014, 2016; Luo et al., 2018). Each node

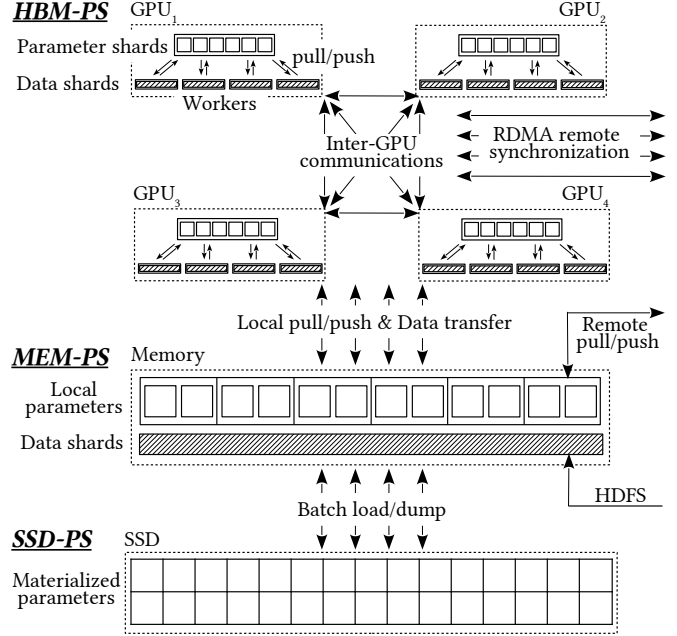


Figure 2. Hierarchical parameter server architecture.

is responsible to process its own training batches—different nodes receive different training data from HDFS. Then, each node identifies the union of the referenced parameters in the current received batch and pulls these parameters from the local MEM-PS/SSD-PS (line 3) and the remote MEM-PS (line 4). The local MEM-PS loads the local parameters stored on local SSD-PS into the memory and requests other nodes for the remote parameters through the network. After all the referenced parameters are loaded in the memory, these parameters are partitioned and transferred to the HBM-PS in GPUs. In order to effectively utilize the limited GPU memory, the parameters are partitioned in a non-overlapped fashion—one parameter is stored only in one GPU. When a worker thread in a GPU requires the parameter on another GPU, it directly fetches the parameter from the remote GPU and pushes the updates back to the remote GPU through high-speed inter-GPU hardware connection NVLink (Foley & Danskin, 2017). In addition, the data batch is sharded into multiple mini-batches and sent to each GPU worker thread (line 5-10). Many recent machine learning system studies (Ho et al., 2013; Chilimbi et al., 2014; Cui et al., 2016; Alistarh et al., 2018) suggest that the parameter staleness shared among workers in data-parallel systems leads to slower convergence. In our proposed system, a mini-batch contains thousands of examples. One GPU worker thread is responsible to process a few thousand mini-batches. An inter-GPU parameter synchronization is performed after each mini-batch is processed to eliminate the parameter staleness. Each training worker pulls the required parameters of its corresponding mini-batch from the HBM-PS (line 12), performs forward and backward propagation to update the parameters (line 13), and interacts with the HBM-PS to up-



**Algorithm 1** Distributed Hierarchical Parameter Server Training Workflow.

---

```

1. while not converged do
2.    $batch \leftarrow get\_batch\_from\_HDFS()$ 
3.    $working \leftarrow pull\_local\_MEM\_PS\_and\_SSD\_PS(batch)$ 
4.    $working \leftarrow working \cup pull\_remote\_MEM\_PS(batch)$ 
5.    $minibatches \leftarrow shard\_batch(batch, \#GPU, \#minibatch)$ 
6.    $partitions \leftarrow map\_parameters\_to\_GPUs(working, \#GPU)$ 
7.   for  $i \leftarrow 1$  to  $\#GPU$  do
8.      $transfer\_to\_GPU(i, minibatches_i)$ 
9.      $insert\_into\_hashtable(i, partitions_i)$ 
10.  end for
11.  for  $j \leftarrow 1$  to  $\#minibatch$  do
12.     $pull\_HBM\_PS\_kernel(minibatch_j)$ 
13.     $\Delta_j \leftarrow train\_mini\_batch\_kernel(j)$ 
14.     $push\_parameters\_updates\_back\_kernel(\Delta_j)$ 
15.  end for
16.   $\Delta \leftarrow pull\_updates\_HBM\_PS()$ 
17.   $parameters\_to\_dump \leftarrow update\_local\_cache()$ 
18.   $push\_local\_SSD\_PS(parameters\_to\_dump)$ 
19. end while

```

---

date the referenced parameters on other GPUs (line 14). The MEM-PS collects the updates from the HBM-PS (line 16) and dumps infrequently used parameters to the SSD-PS when the MEM-PS does not have sufficient memory (line 17-18). An example for Algorithm 1 is shown in Appendix A. HBM-PS, MEM-PS, and SSD-PS communicate in a hierarchical storage fashion. The upper-level module acts as a high-speed cache of the lower-level module.

**HBM-PS.** HBM-PS is distributed in the High-Bandwidth Memory (HBM) across multiple GPUs. Comparing with the conventional distributed parameter servers, workers in GPUs can directly request and update parameters in the HBM without transferring data between GPU memory and CPU memory. Thus the training throughput is significantly increased. However, the HBM capacity is limited and pricey. In order to fit all the terabyte-scale parameters into HBMs, we have to maintain a distributed computing cluster with hundreds of GPUs—it is not only expensive but also inefficient, because excessive inter-GPU and inter-node communications are required to request and update the parameters. Therefore, our proposed hierarchical architecture leverages memory and SSD to store the massive model parameters.

**MEM-PS.** The MEM-PS pulls the referenced parameters from remote nodes to prepare the data for the training in GPUs. After the GPU workers complete the forward/backward propagation, the MEM-PS retrieves the updates from the GPUs, applies the changes, and materializes the updated parameters in the SSDs. The MEM-PS also caches the frequently used parameters to reduce SSD I/Os.

**SSD-PS.** We build the SSD-PS to store the materialized parameters and provide efficient parameter loading and updating. Contiguous addresses readings and writings on SSDs have better I/O bandwidth than random disk accesses. Thus, the SSD-PS organizes the parameters in files. The updated parameters in the memory are written to SSDs in batches as new files. A file compaction thread runs in the background to regularly merge files that contain a large proportion of stale parameters into new files.

**4-stage pipeline.** The training workflow majorly involves 4 time-consuming tasks: data transferring, parameter partitioning, materialized data loading/dumping and neural network training. The 4 tasks correspond to independent hardware resources: network, CPU, SSD and GPU, respectively. We build a 4-stage pipeline to hide the latency of those tasks by maintaining a prefetch queue for each stage. A worker thread is created for each stage—it extracts jobs from the prefetch queue and feeds the corresponding hardware resource. After that, the worker thread pushes the processed results into the prefetch queue of the next stage. Especially, the worker thread stalls when the prefetch queue of the next stage is full—the next stage has already obtained too many unprocessed jobs. The capacity of the prefetch queue is preset according to the execution time of each stage. A detailed explanation is shown in Appendix B.

## 4 HBM-PS

The HBM-PS stores the working parameters in GPUs and provides efficient accesses. In this section, we introduce the multi-GPU distributed hash table and GPU communications. Additional details of HBM-PS are described in Appendix C.

### 4.1 Multi-GPU Distributed Hash Table

Multi-GPU distributed hash table manages the GPU HBMs on the same node and provides a unified interface to the GPU worker threads. All referenced parameters of the current training batch are partitioned and inserted into the local hash table of each GPU. Operations such as *insert*, *get* and *accumulate* are provided to interact with the hash table.

**Local GPU hash table.** Each GPU maintains its own local hash table in the HBM. In this paper, we adopt the `concurrent_unordered_map` of cuDF library<sup>1</sup> as the hash table. The hash table implementation uses the open-addressing method to resolve hash collision. In order to avoid the inefficient dynamic memory allocation in GPUs, we fix the hash table capacity when we construct the hash table—we know the number of parameters (key-value pairs) to be stored on each GPU. For parallel hash table updates, the data integrity is guaranteed by GPU atomic operations.

**Operations.** The multi-GPU distributed hash table provides

<sup>1</sup><https://github.com/rapidsai/cudf>

**Algorithm 2** Distributed GPU hash table accumulate.**Input:** A collections of key-value pairs:  $(keys, vals)$ .

```

1. switch_to_GPU(get_resource_owner(keys, vals))
2. partitioned  $\leftarrow$  parallel_partition(keys, vals)
3. for  $i \leftarrow 1$  to  $\#GPU$  do
4.   if partitioned $i$   $\neq \emptyset$  then
5.     async_send( $i$ , partitioned $i$ )
6.   end if
7. end for
8. wait_send_destination()
9. for  $i \leftarrow 1$  to  $\#GPU$  do
10.  switch_to_GPU( $i$ )
11.  hash_tables $i$ .async_accum(partitioned $i$ )
12. end for

```

operations such as *insert*, *get* and *accumulate*. Here we introduce the *accumulate* operation that updates the parameters during neural network training in detail. The *insert* and *get* operations implement the pull and push interfaces to interact with the HBM-PS. Their workflows are similar to the *accumulate*—their implementation details are omitted due to the page limit.

The *accumulate* operation takes a collection of key-value pairs as input and accumulates the values onto the parameters referenced by the corresponding keys. After each back propagation is finished in the neural network training, *accumulate* operations are called to update the parameters.

Algorithm 2 illustrates the *accumulate* operation workflow. The key-value pairs are passed in a zero-copy fashion—these pairs are represented as the addresses of two arrays—*keys* and *vals*—in the GPU HBM. We first switch to the GPU that owns the memory of the input key-value pairs (line 1). Then we parallelly partition them according to the partition policy. The partitioned results for each GPU is stored at *partitioned* (line 2). After that, we send each partitioned key-value pairs to their corresponding GPUs (line 3-7). The send operation is executed asynchronously through NVLink—we can send data to other GPUs simultaneously. Finally, we iterate over all GPUs and apply the actual accumulation on the local hash table of each GPU asynchronously (line 9-12).

## 4.2 GPU Communication

Since each node processes its own input training batch, these batches may share some working parameters—the shared parameters are referenced and updated by multiple nodes. The multi-GPU distributed hash table stores the working parameters across all GPUs in the same node. We have to synchronize the parameters on different nodes to guarantee the model convergence (Ho et al., 2013; Chilimbi et al., 2014; Cui et al., 2016; Alistarh et al., 2018).

Physically, GPU communication is performed through

Remote Direct Memory Access (RDMA) (Potluri et al., 2013) hardware. RDMA enables zero-copy network communication—it allows the network card to transfer data from a device memory directly to another device memory without copying data between the device memory and the data buffers in the operating system. Our RDMA hardware design eliminates the involvement of the CPU and memory.

Logically, the inter-node GPU parameter synchronization requires an all-reduce communication—each GPU needs to receive all parameter updates from other GPUs and then performs a reduction to accumulate these updates. Appendix C.3 illustrates the implementation details.

## 5 MEM-PS

The MEM-PS identifies the referenced parameters from the input and communicates with the local SSD-PS and remote MEM-PS to gather the required parameters. Meanwhile, a parameter cache is maintained to reduce SSD I/Os. More implementation details are in Appendix D.

**Prepare parameters.** As mentioned in the distributed architecture design (Figure 2), each node only maintains a shard of parameters—parameters are partitioned according to a pre-defined parameter-to-node mapping. We leverage the modulo hashing as the partition scheme in a similar manner to the GPU parameter partition policy discussed in Section 4.1. The MEM-PS partitions the referenced parameters according to their keys. For the remote parameters that belong to a remote node, the MEM-PS pulls them from other MEM-PS through the network. For the local parameters, the MEM-PS reads the SSDs to fetch the parameters. In order to avoid excessive SSD I/Os, an in-memory cache is maintained to store the recently/frequently used parameters and to buffer the working parameters.

**Update parameters.** The MEM-PS pulls the updated parameters from the HBM-PS after its GPUs complete the current batch and applies these changes in the memory (we pin the working parameters in the memory). Note that we do not need to push the changes of the remote parameters to other MEM-PS, because the remote parameters are synchronized across GPUs in HBM-PS—the remote node MEM-PS can pull the updates from its own GPUs.

## 6 SSD-PS

The SSD-PS aims to maintain the materialized out-of-main-memory parameters efficiently on SSDs.

**Parameter files.** We organize the model parameters in the file-level granularity—each file contains a collection of parameters. A parameter-to-file mapping is maintained in the main memory. A file descriptor occupies much less memory than the parameter value—a file descriptor can be

represented as couples of bytes while a parameter value consists of multiple numeric attributes. In addition, a node only contains a shard of the parameters. Thus, we can assume the mapping can fit in the memory of a single node.

**Parameter I/O.** We consider a parameter file as an SSD I/O unit. The SSD-PS gathers requested parameter keys and reads an entire parameter file when it contains requested parameters. On the other hand, parameters evicted from the HBM-PS cache are dumped onto SSDs. It is impractical to locate these parameters and perform in-place updates inside the original file because it randomly writes the disk. Instead, our SSD-PS chunks these updated parameters into files and writes them as new files on SSDs—data are sequentially written onto the disk. After the files are written, we update the parameter-to-file mapping of these parameters. The older versions of the parameters stored in the previous files become stale—these older values will not be used since the mapping is updated. The SSD usage hikes as we keep creating files on SSDs. A file compaction operation is performed regularly to reduce the disk usage—many old files containing a large proportion of stale values can be merged into new files. Appendix E presents the implementation of parameter loading/dumping and file compaction operations.

## 7 EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the overall performance—as well as the impact of optimizations—of the proposed system. Specifically, the experiments are targeted to answer the following questions:

- How does the proposed hierarchical GPU parameter server compare with the MPI cluster solution?
- How is the execution time of each training stage?
- How does the time distribute in each component?
- How does the proposed system scale?

**System.** We execute the distributed hierarchical GPU parameter server experiments on 4 GPU computing nodes. Each node has 8 cutting-edge 32 GB HBM GPUs, server-grade CPUs with 48 cores (96 threads),  $\sim 1$  TB of memory,  $\sim 20$  TB RAID-0 NVMe SSDs and a 100 Gb RDMA network adaptor. The nodes in the MPI cluster for the baseline comparison are maintained in the same data center. CPUs in the MPI cluster have similar performance specifications as the ones in the GPU computing nodes. All nodes are inter-connected through a high-speed Ethernet switch. The hardware and maintenance cost of 1 GPU node roughly equals to the cost of 10 CPU-only MPI nodes.

**Models.** We use 5 CTR prediction models in real-world online sponsor advertising applications to investigate the effectiveness of our proposed system. Table 3 illustrates the specification of these models. The number of sparse param-

Table 3. Model specifications.

	#Non-zeros	#Sparse	#Dense	Size (GB)	MPI
A	100	$8 \times 10^9$	$7 \times 10^5$	300	100
B	100	$2 \times 10^{10}$	$2 \times 10^4$	600	80
C	500	$6 \times 10^{10}$	$2 \times 10^6$	2,000	75
D	500	$1 \times 10^{11}$	$4 \times 10^6$	6,000	150
E	500	$2 \times 10^{11}$	$7 \times 10^6$	10,000	128

eters of each model varies from  $8 \times 10^9$  (model A) to  $10^{11}$  (model E). The number of dense parameters is 4 – 5 orders of magnitude less than the number of sparse parameters for all models. The size of the smallest model is 300 GB, while the largest model has 10 TB parameters. The MPI column shows the number of CPU-only nodes we used to train each model in the MPI cluster for real-world ads products. Due to different training speed requirements for these products, the number of MPI cluster nodes varies from 75 to 150.

**Data.** We collect user click history logs from our search engine as the training dataset. The datasets are chunked into batches—each batch contains  $\sim 4 \times 10^6$  examples. The trained CTR models are evaluated over the production environment of our search engine in an A/B testing manner.

### 7.1 Comparison with MPI Solution

We evaluate the performance of the proposed system on the training execution time and the prediction accuracy. Our proposed system runs on 4 GPU computing nodes. We take an MPI cluster training framework in the production environment as the baseline. The baseline solution employs a distributed parameter server that shards and keeps the massive parameters in the memory across all the nodes.

**Training time.** Figure 3(a) depicts the total training execution time of the 4-node hierarchical parameter server (HPS-4) and the MPI training solution (MPI-cluster). HPS-4 outperforms MPI-cluster in all 5 models. The first row of Table 4 shows the speedup over the MPI solution. The HPS-4 is 2.4-4.8X faster than the MPI solution. Note that the hardware and maintenance cost of 1 GPU node is roughly equivalent to 10 CPU-only nodes in the MPI cluster. The 4-GPU-node setting is much cheaper than the 75-150 nodes in the MPI cluster. The second row of the table illustrates the cost-normalized speedup of our proposed system. The cost-normalized speedup is computed as:  $speedup/4/10 \times \#MPI$ . The price-performance ratio of our proposed system is 4.4-9.0X better than the MPI cluster.

Table 4. The speedup over the MPI-cluster solution and the normalized speedup at the same hardware and maintenance cost.

	A	B	C	D	E
Speedup over MPI-cluster	1.8	2.7	4.8	2.2	2.6
Cost-normalized speedup	4.4	5.4	9.0	8.4	8.3

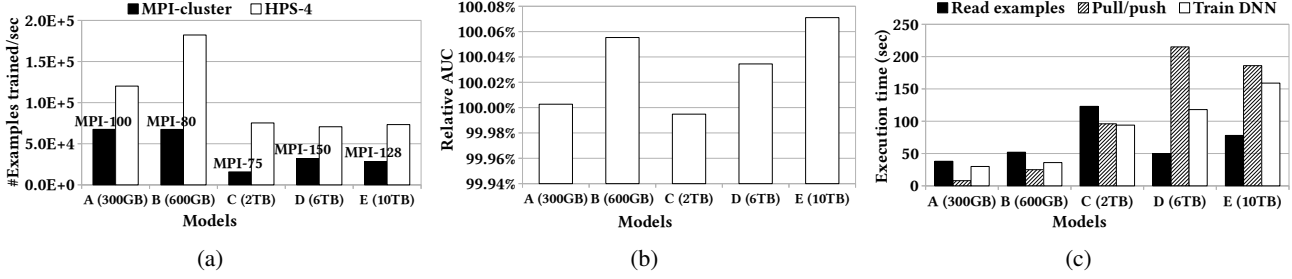


Figure 3. (a) Total execution time, (b) AUC accuracy, and (c) Execution time distribution.

**Accuracy.** We take Area Under the Curve (Huang & Ling, 2005) (AUC) as the quality measure of our trained models. Figure 3(b) shows the relative AUC of HPS-4 (relative to the MPI-cluster method). The AUC of the MPI-cluster solution is 100%. The AUCs of both trained models are tested online for 1 day in real online ads products. Since the CTR prediction accuracy is crucial to the revenue, we have to ensure all our optimizations are loss-less. For Model C, the relative AUC loss of HPS-4 is less than 0.01%. For other 4 models, HPS-4 has even slightly better accuracy than the MPI-128. Our hierarchical parameter server runs on fewer nodes—fewer stale parameters are used than the case on the MPI cluster. Therefore, it is possible to have a better AUC than the MPI solution. Overall, the relative differences of all 5 models are within 0.1%. We can conclude that our hierarchical parameter server training is loss-less.

## 7.2 Time distribution

The time distribution of each abstract execution stage is presented in Figure 3(c). As illustrated in the training workflow (Algorithm 1), *Read examples* corresponds to the operation that reads and extracts the data from HDFS; *Pull/push* is the execution time of gathering and updating the working parameters in the MEM-PS and SSD-PS; and *Train DNN* is the execution time of the inter-GPU communication and the deep neural network training in the HBM-PS. Since these stages are paralleled in the pipeline, the overall execution time for each batch is dominated by the slowest stage. For the two small models (Model A and B), the *Read examples* stage that relies on the HDFS throughput becomes the bottleneck. Because the models are small and fit in the memory, the sparse parameter pulling and pushing operations are faster than the HDFS I/Os. When the number of sparse parameters grows, the *Pull/push* operations catch up with the example reading (Model C) and become the dominant stage in Model D and E.

## 7.3 HBM-PS

The execution time distribution of HBM-PS operations is shown in Figure 4(a). The pull/push operations of HBM-

PS depend on the number of non-zero values of training inputs. While the training operation (forward/backward propagation) correlates to the number of dense parameters of the deep model. The execution time of these operations shows the same trend in the model specification (Table 3). Model A and Model B have around 100 non-zero input features per example, the pull/push HBM-PS operations are faster than the ones of Model C, D and E. Model E has the largest number of dense parameters. Thus its training operation costs most of the execution time in HBM-PS.

## 7.4 MEM-PS

**Local/remote parameters.** Figure 4(b) illustrates the execution time of pulling local/remote parameters in the MEM-PS for Model E over 1, 2 and 4 GPU nodes. The results of other models show a similar trend. The remote parameter pulling operation is not applicable when we only have 1 GPU node—all the parameters are stored in the same node. When we deploy the proposed system in distributed environments with 2 and 4 nodes, the local and remote pulling operations are paralleled. The overall execution time is determined by the slower operation. We can observe that the overall time of pulling MEM-PS parameters does not hike much when inter-node communications are involved.

**Cache performance.** The cache hit rate for Model E is illustrated in Figure 4(c). In the beginning of the training, the cache hit rate is low because we start from a cold cache—no data are cached. We can observe that the cache hit rate increases steeply during the training of the first 10 batches. After 40 training batches, the cache hit rate goes to 46% and becomes stable in the following training—the frequently visited parameters are captured by our cache policy.

## 7.5 SSD-PS

Figure 5(a) presents the SSD I/O time for Model E. The SSD disk usage threshold is reached and the parameter file compaction operation is involved from the 54<sup>th</sup> batch—the SSD I/O time hikes. The regular file merging in the compaction operation causes the I/O performance fluctuation.



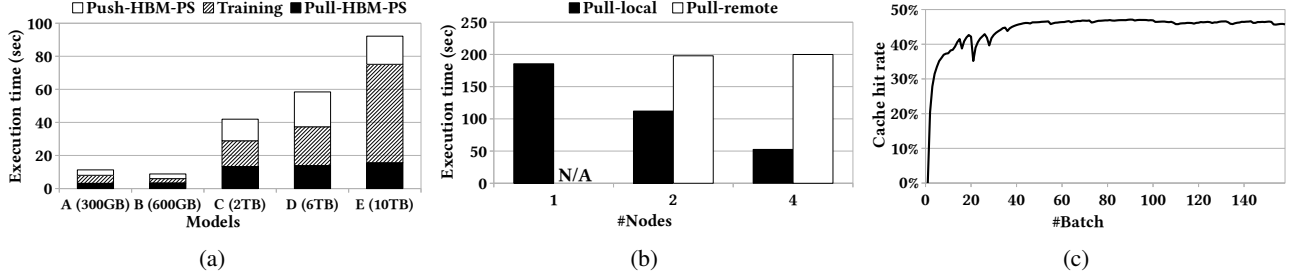


Figure 4. (a) Time distribution in HBM-PS; Model E: (b) Time Distribution in MEM-PS and (c) Cache hit rate.

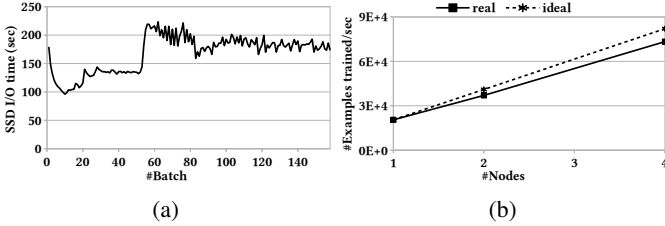


Figure 5. Model E: (a) SSD-PS I/O time; (b) Speedup

## 7.6 Scalability.

The training throughput (number of examples trained per second) speedup is depicted in Figure 5(b) over 1, 2, and 4 nodes. The dashed line represents the ideal linear speedup. The reason for the (slightly) sub-linear speedup is that more network communications are incurred when we use more nodes. A speedup of 3.57 out of 4 is obtained for 4 nodes.

## 7.7 Discussion

Based on the results shown above, we can answer the questions driving the experimental evaluation. The 4-node distributed hierarchical GPU parameter server is 2.4-4.8X faster than the MPI solution in the production environment. The cost of 4 GPU nodes is much less than the cost of maintaining 75-150 CPU nodes in the MPI cluster. After normalizing the execution time by the hardware and maintenance cost, the price-performance ratio of our proposed system is 4.4-9.0X better than the MPI solution. Besides the training speed, the relative accuracy difference to the MPI solution is within 0.1%—the training in our proposed system is loss-less. The proposed pipeline parallels all training stages. The overall execution time for each batch is dominated by the HDFS when the model is small. When the number of sparse parameters grows to 6 and 10 TB, the sparse parameter pulling and pushing operations become the dominant factor. The execution time in the HBM-PS depends on the model specifications. A larger number of non-zero values of training inputs yields longer execution time for pulling and pushing parameters in the HBM-PS. The training forward and backward propagation time depend on the size of

fully-connected layers in the deep model. The execution time of pulling working parameters in the MEM-PS is almost independent of the number of nodes. Although the local data required to load from SSDs are reduced when we have multiple nodes, more parameter requests from remote nodes have to be processed. In expectation, the average number of parameters to load from SSDs stays the same in multi-node cases. The SSD-PS performance has slight fluctuations when the parameter file compaction operation involves in. Our proposed system scales well in distributed environments—a training throughput speedup of 3.57 out of 4 (the ideal speedup) is obtained for 4 nodes.

## 8 RELATED WORK

In this section, we discuss relevant work from CTR prediction models and parameter servers. Additional related work about in-memory cache management systems and key-value stores on SSDs are discussed in Appendix F.

**CTR prediction models.** The large scale logistic regression model with careful feature engineering used to dominate the CTR prediction strategies (Edelman et al., 2007; Graepel et al., 2010). Recently, deep neural networks with embedding layers are applied in the CTR prediction problem and obtain significant improvements. Generally, the sparse input features are converted into dense vectors through embedding layers, and then feed into neural components to expose the correlation/interaction among them. The differences among these models lie in the neural components above the embedding layer. For example, Deep Crossing (Shan et al., 2016), Wide&Deep Learning (Cheng et al., 2016), YouTube Recommendation CTR model (Covington et al., 2016) and Deep Interest Network (DIN) (Zhou et al., 2018) design special fully-connected layers for corresponding tasks to capture the latent interactions among features; Product-based Neural Network (PNN) (Qu et al., 2016) employs a product layer to capture high-order feature interactions; DeepFM (Guo et al., 2017) and xDeepFM (Lian et al., 2018) use factorization machines (FM) to model both low-order and high-order feature correlations. Through this approach, the correlations between sparse features are automatically exploited.

The successes of these systems have proven that deep learning with sparse input is an effective approach for real-world commercial advertisement and recommendation systems. However, when the scale of the problem becomes extremely huge, for example, with  $10^{11}$  dimensional sparse data and trillions of samples, the architecture of these models must be carefully designed to cope with the challenges mentioned in above sections. This paper is our attempts to solving these challenges using an SSD based, three-layer distributed GPU training architecture. We believe our solution greatly enhances the feasibility of the methodology of deep learning with sparse input in much larger scale problems.

**Parameter servers.** One important research direction is the parallel mechanism among the computational nodes and server nodes. One such topic is the synchronization pattern of local updates of parameters, where three typical types of synchronization patterns are proposed: 1) Bulk Synchronous Parallel (BSP) (Valiant, 1990), which strictly synchronizes all local updates from all computing nodes in each iteration, and thus the learning procedure acts as the same logic as that of learning in a single machine. 2) Stale Synchronous Parallel (SSP) (Ho et al., 2013), which allows the fastest computing node to run a certain number of iterations ahead of other nodes. 3) Asynchronous Parallel (ASP), which does not need any synchronization among the updates of computing nodes. Instead of using the same parallel strategy throughout the training procedure, FlexPS (Huang et al., 2018) proposed a flexible parallelism control for multi-stage learning algorithms such as SVRG.

Another direction is to design efficient parameter server architecture for different hardware/network situations. Communication is a major bottleneck in parallel training. Poseidon (Zhang et al., 2017) develops two strategies *wait-free back-propagation* and *hybrid communication* to exploit the independency between parameters of layered model structures in deep neural networks and conquer communication bottleneck; while Gaia (Hsieh et al., 2017) even considered the problem of running learning program on geo-distributed clusters that communicate over WANs. Another challenge is the big model size for GPU platforms. When the model becomes bigger, the limited GPU memory cannot hold the entire model. GeePS (Cui et al., 2016) builds a two-layer architecture to cope with the problems in this situation, including data movement overheads, GPU stalls, and limited GPU memory, where the Memory layer holds the entire model in memory, and the GPU layer fetch the required part of the model from the memory layer, do the computation and rewrite the updates to the memory layer. Our proposed system constructs a distributed hash table across GPU HBMs that enables direct GPU peer-to-peer communications. It reduces the excessive CPU-GPU data movement and synchronization overhead in GeePS.

However, when the model size becomes even larger, the parameters cannot even be stored in the CPU main memory of the cluster. Our solution to the challenge is a three-layer architecture that consists of SSD-PS, MEM-PS layer, and HBM-PS. The new architecture results in new challenges such as GPU I/O problems and we develop strategies to optimize the performance. Our architecture can efficiently train a 10 TB model, which greatly enhances the capability of GPU clusters to cope with massive scale models.

## 9 CONCLUSIONS

Since 2013, Baidu Search Ads (a.k.a. “Phoenix Nest”) has been successfully using ultra-high dimensional input data and ultra-large-scale deep neural networks for training CTR prediction models. In this paper, we introduce the architecture of a distributed hierarchical GPU parameter server for massive scale deep learning ads systems. As discussed in Section 2, model hashing techniques such as “one-permutation + one sign random projection” are not fully applicable to the accuracy-crucial ads industry applications. The deep learning model parameter size can become huge and cannot fit in the CPU/GPU memory. The proposed hierarchical design employs the main memory and SSDs to, respectively, store the out-of-GPU-memory and out-of-main-memory parameters of massive scale neural networks.

We perform an extensive set of experiments on 5 CTR prediction models in real-world online sponsor advertising applications. The results confirm the effectiveness and the scalability of the proposed system. The 4-node distributed hierarchical GPU parameter server is 2.4-4.8X faster than the MPI solution in the production environment. For example, a 4-node hierarchical GPU parameter server can train a model more than 2X faster than a 150-node in-memory distributed parameter server in an MPI cluster on Model D. The cost of 4 GPU nodes is much less than the cost of maintaining an MPI cluster of 75-150 CPU nodes. After normalizing the execution time by the hardware cost and maintenance expenditure, the price-performance ratio of this proposed system is 4.4-9.0X better than the previous MPI solution.

The system described in this paper is being integrated with the PaddlePaddle deep learning platform (<https://www.paddlepaddle.org.cn>) to become the “PaddleBox”.

## ACKNOWLEDGEMENT

The system described in this paper represents the substantial effort in the past years involving several major product teams at Baidu Inc. We could not include all the names we should acknowledge but only a few: Xuewu Jiao, Quanxiang Jia, Lian Zhao, Lin Liu, Jiajun Zhang, Yue Wang, Anlong Qi.

## REFERENCES

- Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen M. Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. *World Wide Web Journal*, 1(1), 1996.
- Dan Alistarh, Christopher De Sa, and Nikola Konstantinov. The convergence of stochastic gradient descent in asynchronous shared memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 169–178, New York, NY, 2018.
- Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 433–448, San Jose, CA, 2010.
- David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 1–14, Big Sky, MT, 2009.
- Andrei Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.
- Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *1st USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, 1997.
- Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
- Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, (DLRS@RecSys)*, pp. 7–10, Boston, MA, 2016.
- Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 571–582, Broomfield, CO, 2014.
- Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.
- Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithm*, 55(1):58–75, 2005.
- Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys)*, pp. 191–198, Boston, MA, 2016.
- Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC)*, pp. 37–48, Philadelphia, PA, 2014.
- Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, pp. 4:1–4:16, London, UK, 2016.
- Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB)*, pp. 330–341, Mumbai (Bombay), India, 1996.
- Biplob K. Debnath, Sudipta Sengupta, and Jin Li. Flash-store: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.
- Biplob K. Debnath, Sudipta Sengupta, and Jin Li. Skimpys-tash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 25–36, Athens, Greece, 2011.
- Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *American economic review*, 97(1):242–259, 2007.
- Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922*, 2018.
- Daniel C. Fain and Jan O. Pedersen. Sponsored search: A brief history. *Bulletin of the American Society for Information Science and Technology*, 32(2):12–13, 2006.
- Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. MOBIUS: towards the next generation of query-ad matching in baidu’s sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pp. 2509–2517, Anchorage, AK, 2019.

- Denis Foley and John Danskin. Ultra-performance pascal GPU and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.
- Thore Graepel, Joaquin Quiñero Candela, Thomas Borchert, and Ralf Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft’s bing search engine. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 13–20, Haifa, Israel, 2010.
- Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for ctr prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1725–1731, Melbourne, Australia, 2017.
- Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1223–1231, Lake Tahoe, NV, 2013.
- Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 629–647, Boston, MA, 2017.
- Jin Huang and Charles X Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(3):299–310, 2005.
- Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *PVLDB*, 11(5):566–579, 2018.
- Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pp. 489–504, Houston, TX, 2018.
- Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.
- Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 583–598, Broomfield, CO, 2014.
- Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. Hashing algorithms for large-scale learning. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2672–2680, Granada, Spain, 2011.
- Ping Li, Art B Owen, and Cun-Hui Zhang. One permutation hashing. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 3122–3130, Lake Tahoe, NV, 2012.
- Ping Li, Xiaoyun Li, and Cun-Hui Zhang. Re-randomized densification for one permutation hashing and bin-wise consistent weighted sampling. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, 2019.
- Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pp. 1754–1763, London, UK, 2018.
- Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 1–13, Cascais, Portugal, 2011.
- Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.
- Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 41–54, Carlsbad, CA, 2018.
- Shamkant Navathe et al. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.
- Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 297–306, Washington, DC, 1993.



- Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient inter-node mpi communication using gpudirect RDMA for infiniband clusters with nvidia GPUs. In *42nd International Conference on Parallel Processing (ICPP)*, pp. 80–89, Lyon, France, 2013.
- Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-based neural networks for user response prediction. In *IEEE 16th International Conference on Data Mining (ICDM)*, pp. 1149–1154, Barcelona, Spain, 2016.
- Ying Shan, T. Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and J. C. Mao. Deep crossing: Web-scale modeling without manually crafted combinatorial features. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 255–262, San Francisco, CA, 2016.
- Anshumali Shrivastava and Ping Li. Improved asymmetric locality sensitive hashing (ALSH) for maximum inner product search (MIPS). In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 812–821, Amsterdam, The Netherlands, 2015.
- Leonid B. Sokolinsky. Lfu-k: An effective buffer management replacement algorithm. In *9th International Conference on Database Systems for Advances Applications (DASFAA)*, pp. 670–681, Jeju Island, Korea, 2004.
- Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pp. 1113–1120, Montreal, Canada, 2009.
- Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks*, 29(8-13):977–986, 1997.
- Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pp. 181–193, Santa Clara, CA, 2017.
- Weijie Zhao, Yu Cheng, and Florin Rusu. Vertical partitioning for query processing over raw data. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management, (SSDBM)*, pp. 15:1–15:12, La Jolla, CA, 2015.
- Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 319–328, Beijing, China, 2019.
- Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pp. 1059–1068, London, UK, 2018.

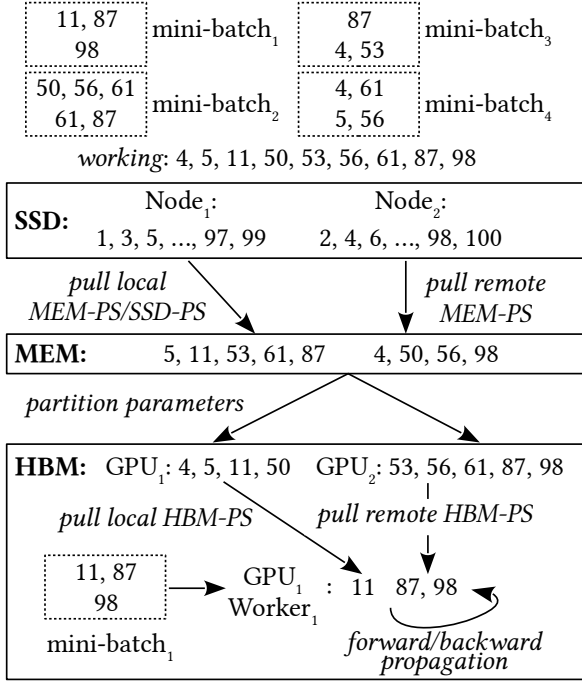


Figure 6. An example for Algorithm 1.

## A HIERARCHICAL PARAMETER SERVER WORKFLOW EXAMPLE

**Example.** Figure 6 depicts an example for the training workflow (Algorithm 1). Consider now we are at *node*<sub>1</sub>. An input batch is streamed from HDFS and is divided into 4 mini-batches. The working parameters of the current batch are: 4, 5, 11, 50, 53, 56, 61, 87, 98. Parameters are sharded and stored on the SSDs of each node. We have 2 nodes and shard the parameters in a round-robin method in this example—*node*<sub>1</sub> stores the parameters with odd keys while *node*<sub>2</sub> stores the ones with even keys. Here we have 100 total parameters in this example—there are  $10^{11}$  parameters in real-world large-scale deep learning models. 5, 11, 53, 61, 87 are stored on the local node—*node*<sub>1</sub>. We pull these parameters from the local MEM-PS (for the cached parameters) and the local SSD-PS. For the parameters that stored on other nodes—4, 50, 56, 98, we pull these parameters from the MEM-PS on *node*<sub>2</sub> through the network. The MEM-PS on *node*<sub>2</sub> interacts with its memory cache and its local SSD-PS to load the requested parameters. Now all the working parameters are retrieved and are stored in the memory of *node*<sub>1</sub>. Here we have 2 GPUs on *node*<sub>1</sub>. The working parameters are partitioned and transferred to GPU HBMs. In this example, *GPU*<sub>1</sub> obtains the parameters whose keys are less than or equal to 50—4, 5, 11, 50, and *GPU*<sub>2</sub> takes 53, 56, 61, 87, 98. The partition strategy can be any hashing function that maps a parameter key to a GPU id. Consider the *worker*<sub>1</sub> of *GPU*<sub>1</sub> is responsible to process *mini-batch*<sub>1</sub>. *worker*<sub>1</sub> is required to load 11, 87 and 98. Among them,

11 is stored in the HBM of the local GPU—*GPU*<sub>1</sub>. 87 and 98 are pulled from *GPU*<sub>2</sub>. Since the GPUs are connected with high-speed interconnections—NVLink, the inter-GPU data transfer has low-latency and high-bandwidth. After the parameters are ready in the working memory of *worker*<sub>1</sub>, we can perform the neural network forward and backward propagation operations to update the parameters. All the updated parameters are synchronized among all GPUs on all nodes after each mini-batch is finished. When all the mini-batches are finished, the MEM-PS on each node pulls back the updated parameters and materializes them onto SSDs.

## B 4-STAGE PIPELINE EXAMPLE

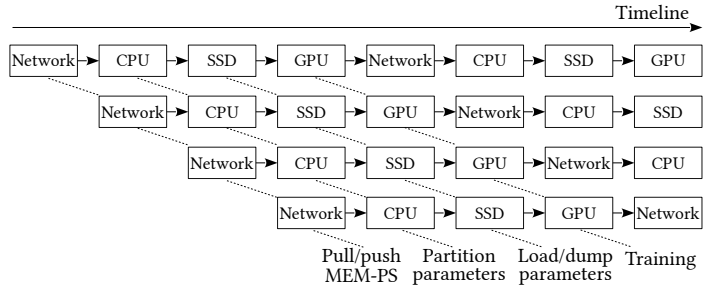


Figure 7. The 4-stage pipeline.

Figure 7 is an illustration of the 4-stage pipeline. For example, when the GPUs are busy training the model, our 4-stage pipeline enables the proposed system to prepare the referenced parameters of the next training batch at the same time: the HBM-PS pulls remote parameters from other nodes and the SSD-PS loads local parameters for the next batch simultaneously. After the training of the current batch is finished, all the required parameters of the next batch are ready to use in the GPU HBM—GPUs are able to train the next batch immediately.

## C HBM-PS IMPLEMENTATION

### C.1 Multi-GPU Distributed Hash Table

**Partition policy.** A partition policy that maps a parameter key to a GPU id is required to partition the parameters. A simple modulo hash function yields a balanced partitioning in general cases, because the features of the input training data are usually distributed randomly. The modulo hash function can be computed efficiently with constant memory space. As a trade-off of the memory footprint, the disadvantage of the simple hash partition policy is that we need to pull parameters from other GPUs if the parameters referenced in a mini-batch are not stored in the local parameter partition. One possible improvement is to group parameters with high co-occurrence together (Eisenman et al., 2018), for example, pre-train a learned hash

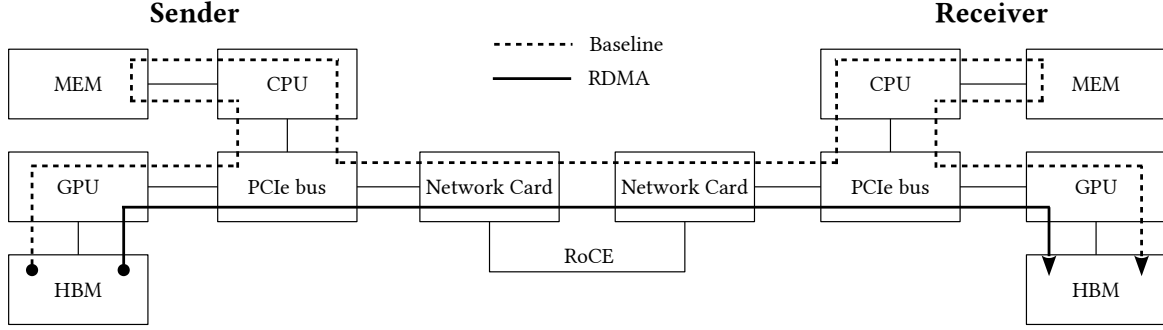


Figure 8. Inter-node RDMA communication.

function (Kraska et al., 2018) to maximize the parameter co-occurrence. It is another research axis—vertical partitioning (Navathe et al., 1984; Zhao et al., 2015) that is beyond the scope of this paper. Generally, no perfect balanced partition solution exists for random inputs. Although the number of pulled parameters is reduced, we still have to pull parameters from almost all other GPUs even with an optimized partition policy. Besides, transferring a large batch of data can better utilize the NVLink bandwidth—the disadvantage of the simple hash function partition policy is reduced.

## C.2 GPU RDMA Communication

**GPU Communication mechanism.** The inter-node GPU communication mechanism is depicted in Figure 8. Two nodes are shown in the figure—the left node is the sender and the right one is the receiver. For each node, the CPU, GPU and network card are connected with a PCIe bus.

We first examine the *baseline* method before we introduce our *RDMA* solution. In the figure, the data flow of the baseline method is represented as the dashed line starting from the sender HBM to the receiver HBM. The CPU calls the GPU driver to copy the data from GPU HBM into the CPU memory. Then, the CPU reads the data in the memory and transmits the data through the network. The transmitted data are stored in the memory of the receiver node. Finally, the receiver CPU transfers the in-memory data to the GPU HBM. In this baseline method, the CPU memory is utilized as a buffer to store the communication data—it incurs unnecessary data copies and CPU consumptions.

Our *RDMA* hardware design eliminates the involvement of the CPU and memory. Its data flow is represented as a solid line in the figure. Remote Direct Memory Access (RDMA) (Potluri et al., 2013) enables zero-copy network communication—it allows the network card to transfer data from a device memory directly to another device memory without copying data between the device memory and the data buffers in the operating system. The RDMA data transfer demands no CPU consumption or context switches. The network protocol—RDMA over Converged Ethernet (RoCE)—

is employed to allow RDMA over the Ethernet network. The sender GPU driver<sup>2</sup> directly streams the data in HBM to the network, while the receiver network card directly stores the collected data into the GPU HBM.

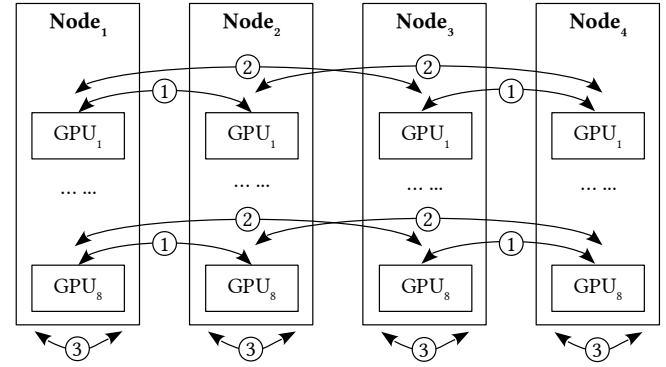


Figure 9. All-reduce communication.

## C.3 Inter-Node GPU Communication

**All-reduce communication.** The parameter synchronization requires an all-reduce communication—each GPU needs to receive all parameter updates from other GPUs and then performs a reduction to accumulate these updates. Figure 9 presents an example communication workflow. 4 nodes are shown in this example. Each node contains 8 GPUs. Initially, the GPUs on *Node1* exchange their parameter updates with their corresponding GPUs on *Node2* (step ①)—i.e., the  $i^{th}$  GPU on *Node1* communicates with the  $i^{th}$  GPU on *Node2*. Meanwhile, the GPUs with the same id on *Node3* and *Node4* share their data with each other. Then, the GPUs on *Node1* perform the communication with the ones on *Node3* (step ②). Likewise, the GPUs on *Node2* and *Node4* perform the same pattern communication in parallel. After these two steps, each GPU on each node has collected all the parameter updates of its corresponding

<sup>2</sup><https://docs.nvidia.com/cuda/gpudirect-rdma/index.html>

GPUs on other nodes. An intra-node GPU tree all-reduce communication<sup>3</sup> is executed to share the data across all 8 GPUs on the same node (step ③). Most of the communications are paralleled— $\log_2 \#nodes$  non-parallel inter-node and  $\log_2 \#GPUs$  intra-node all-reduce communications are required to synchronize the parameters across all nodes.

#### C.4 Dense Parameters

As we discussed in the CTR prediction neural network example (Figure 1), besides the large-scale sparse parameters, there are a small number of dense parameters for the fully-connected layers. For any sparse input, all the dense parameters are referenced and updated. Therefore, we can pin these dense parameters in the HBM of all GPUs at the beginning of the training for better training performance. In extreme cases—we have insufficient HBM memory to replicate the dense parameters, we can shard the dense parameters as the sparse parameters and distribute them across all GPU HBMs. The dense parameters are also synchronized as the sparse ones in the HBM-PS after each mini-batch is processed.

### D MEM-PS IMPLEMENTATION

**Cache policy.** We target to cache the most recently and the most frequently used parameters in the memory to reduce SSD I/Os. In this paper, we leverage a cache eviction policy that combines two cache methods—Least Recently Used (LRU) (O’Neil et al., 1993) and Least Frequently Used (LFU) (Sokolinsky, 2004). Whenever we visit a parameter, we add it into an LRU cache. For the evicted parameters from the LRU cache, we insert them into an LFU cache. The evicted parameters from the LFU cache are collected—we have to flush them into SSDs before releasing their memory. To guarantee the data integrity of our pipeline, we pin the working parameters of the current batch and the pre-fetched parameters of next iterations in the LRU cache—they cannot be evicted from the memory until their batch is completed.

### E SSD-PS IMPLEMENTATION

**Load parameters.** The SSD-PS gathers requested parameter keys from the MEM-PS and looks up the parameter-to-file mapping to locate the parameter files to read. We have to read an entire parameter file when it contains requested parameters—a larger file causes more unnecessary parameter readings. This is a trade-off between the SSD I/O bandwidth and the unnecessary parameter reading—a small-size file cannot fully utilize the SSD I/O bandwidth. We tune the file size to obtain the optimal performance. Figure 10(a) depicts an example of parameter files on SSDs. In

the example, each parameter file can store 3 parameters.

**Dump parameters.** Parameters evicted from the HBM-PS cache are required to be dumped onto SSDs. It is impractical to locate these parameters and perform in-place updates inside the original file—it poorly utilizes the SSD I/O bandwidth because it requires us to randomly write the disk. Instead, our SSD-PS chunks these updated parameters into files and writes them as new files on SSDs—data are sequentially written onto the disk. After the files are written, we update the parameter-to-file mapping of these parameters. The older versions of the parameters stored in the previous files become stale—these older values will not be used since the mapping is updated. In Figure 10(b), we present an example for dumping parameters—1, 2, 4, 6, 8, 9 are updated and dumped to SSD-PS. We chunked them into two files and write these two files onto the SSD—*file<sub>4</sub>* and *file<sub>5</sub>*. The underlined values—the values of the updated parameters in the old files—are stale.

**File compaction.** The SSD usage hikes as we keep creating new files on SSDs. A file compaction operation is performed regularly to reduce the disk usage—many old files containing a large proportion of stale values can be merged into new files. We adopt the leveled compaction algorithm of LevelDB<sup>4</sup> to create a lightweight file merging strategy. A worker thread runs in the background to check the disk usage. When the usage reaches a pre-set threshold, the SSD-PS scans the old parameter files, collects the non-stale parameters, merges them into new files, and erases the old files. The parameter-to-file mapping of the merged parameters is also updated in the file compaction operation. Figure 10(c) illustrates the file compaction effects. Before the compaction (Figure 10(b)), the stale values in *file<sub>1</sub>*, *file<sub>2</sub>* and *file<sub>3</sub>* occupy more than a half of the file capacity. We scan these files, merge the non-stale values into a new file (*file<sub>6</sub>*), and erase these files (*file<sub>1</sub>* and *file<sub>2</sub>*). The compaction operation may merge a large number of files on SSDs. In order to reduce the excessive merging, we set a threshold to limit the number of merged files—we only merge files that contain more than 50% stale parameters. By employing this threshold, we can limit the total SSD space usage—the size of all parameter files will not exceed 2 times (1/50%) of the original non-stale parameter size. Note that we do not need to read the entire file to obtain the proportion of stale parameters—a counter that counts the number of stale parameters is maintained as an auxiliary attribute for each file. When we update the parameter-to-file mapping, we accumulate the counter of the old file it previously maps to.

### F ADDITIONAL RELATED WORK

**In-memory cache management.** Many caching policies have been developed for storage systems, such as the LRU-

<sup>3</sup><https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/usage/operations.html#allreduce>

<sup>4</sup><https://github.com/google/leveldb>



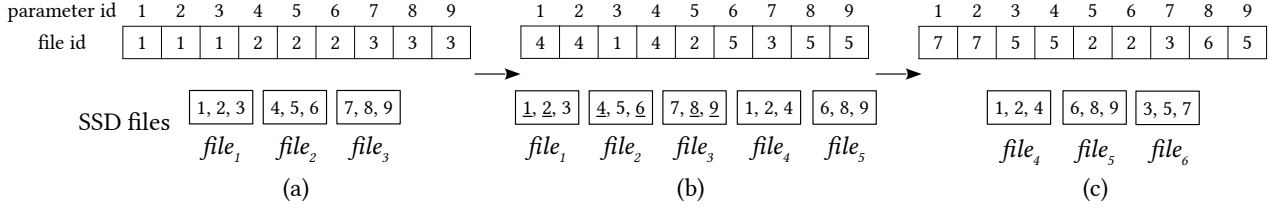


Figure 10. SSD-PS examples: (a) parameter-to-file mapping and parameter files; (b) 1, 2, 4, 8, 9 are updated; (c) a compaction operation.

K (O’Neil et al., 1993), DBMIN (Chou & DeWitt, 1986), LRFU (Lee et al., 2001), and Semantic Caching (Dar et al., 1996). These algorithms evict cache according to a combined weight of recently used time-stamp and frequency. In the web context, there is extensive work developed for variable-size objects. Some of the most well-known algorithms in this space are Lowest-Latency-First (Wooster & Abrams, 1997), LRU-Threshold (Abrams et al., 1996), and Greedy-Dual-Size (Cao & Irani, 1997). Unlike our caching problem, the parameter we tackle with has a fixed size and a clear access pattern in our CTR prediction model training—some parameters are frequently referenced. It is effective to keep those “hot parameters” in the cache by applying an LFU eviction policy. While our additional LRU linked list maintains the parameters referenced in the current pass to accelerate the hash table probing.

**Key-value store for SSDs.** There is a significant amount of work on key-value stores for SSD devices. The major

designs (Andersen et al., 2009; Lim et al., 2011) follow the paradigm that maintains an in-memory hash table and constructs an append-only LSM-tree-like data structure on the SSD for updates. FlashStore (Debnath et al., 2010) optimize the hash function for the in-memory index to compact key memory footprints. SkimpyStash (Debnath et al., 2011) moves the key-value pointers in the hash table onto the SSD. BufferHash (Anand et al., 2010) builds multiple hash tables with Bloom filters for hash table selection. WiscKey (Lu et al., 2017) separates keys and values to minimize read/write amplifications. Our SSD-PS design follows the mainstream paradigm, while it is specialized for our training problem. We do not need to confront the challenges to store general keys and values. The keys are the index of parameters that distributes uniformly. It is unnecessary to employ any sophisticated hashing functions. Also, the values have a known fixed length, the serialized bucket on SSD exactly fits in an SSD block—I/O amplification is minimized.