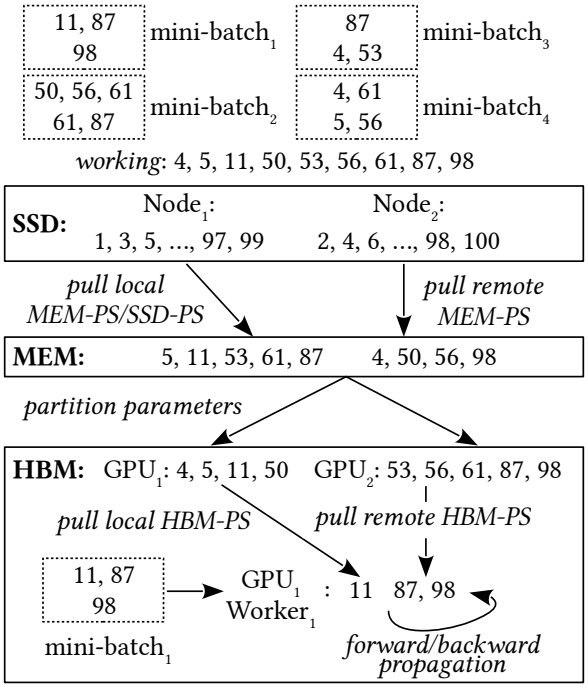

Figure 6. An example for Algorithm 1.

A Hierarchical Parameter Server Workflow Example

Example. Figure 6 depicts an example for the training workflow (Algorithm 1). Consider now we are at $node_1$. 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—$node_1$ stores the parameters with odd keys while $node_2$ 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—$node_1$. 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 $node_2$ through the network. The MEM-PS on $node_2$ 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 $node_1$. Here we have 2 GPUs on $node_1$. The working parameters are partitioned and transferred to GPU HBMs. In this example, GPU_1 obtains the parameters whose keys are less than or equal to 50—$4, 5, 11, 50$, and GPU_2 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 $worker_1$ of GPU_1 is responsible to process $mini\text{-}batch_1$. $worker_1$ is required to load 11, 87 and 98. Among them, 11 is stored in the HBM of the local GPU—GPU_1. 87 and 98 are pulled from GPU_2. 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 $worker_1$, 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

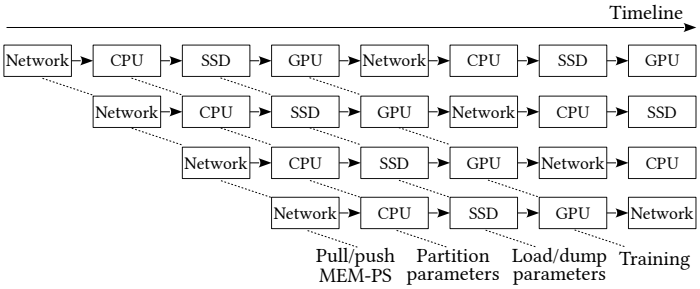

Figure 7. The 4-stage pipeline.

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

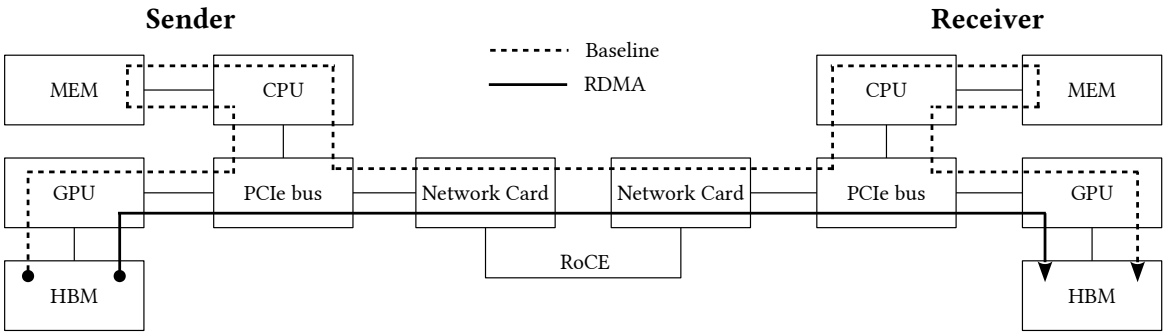

Figure 8. Inter-node RDMA communication.

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[1] directly streams the data in HBM to the network, while the receiver network card directly stores the collected data into the GPU HBM.

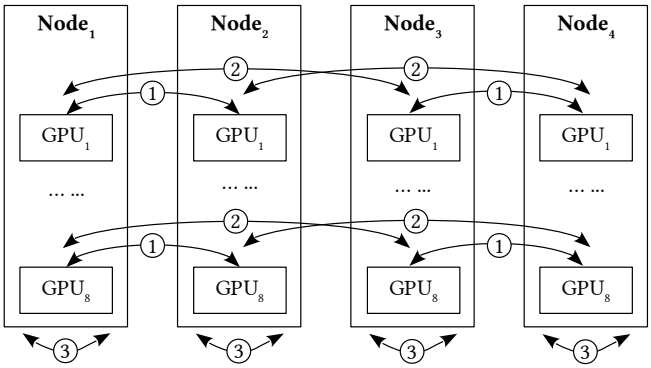

Figure 9. All-reduce communication.

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 $Node_1$ exchange their parameter updates with their corresponding GPUs on $Node_2$ (step ①)—i.e., the i^{th} GPU on $Node_1$ communicates with the i^{th} GPU on $Node_2$. Meanwhile, the GPUs with the same id on $Node_3$ and $Node_4$ share their data with each other. Then, the GPUs on $Node_1$ perform the communication with the ones on $Node_3$ (step ②). Likewise, the GPUs on $Node_2$ and $Node_4$ 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

[1] https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

GPUs on other nodes. An intra-node GPU tree all-reduce communication[2] 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—$file_4$ and $file_5$. 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 [3] 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 $file_1$, $file_2$ and $file_3$ occupy more than a half of the file capacity. We scan these files, merge the non-stale values into a new file ($file_6$), and erase these files ($file_1$ and $file_2$). 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-

[2] https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/usage/operations.html#allreduce

[3] https://github.com/google/leveldb

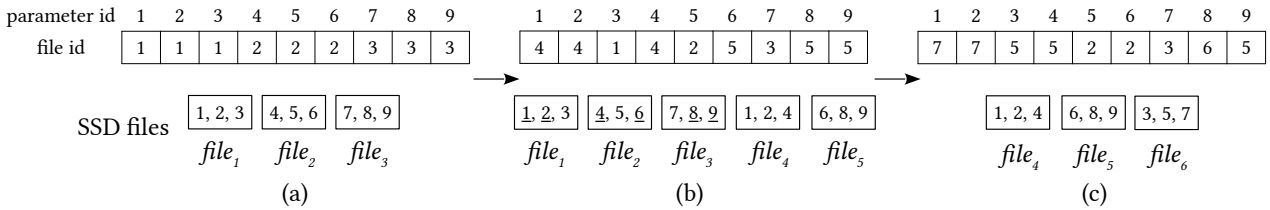

Figure 10. SSD-PS examples: (a) parameter-to-file mapping and parameter files; (b) $1, 2, 4, 8, 9$ are updated; (c) a compaction operation.

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. WisckKey (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.