Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards

Youngeun Kwon
School of Electrical Engineering
KAIST
yekwon@kaist.ac.kr

Minsoo Rhu
School of Electrical Engineering
KAIST
mrhu@kaist.ac.kr

ABSTRACT
Personalized recommendation models (RecSys) are one of the most popular machine learning workload serviced by hyperscalers. A critical challenge of training RecSys is its high memory capacity requirements, reaching hundreds of GBs to TBs of model size. In RecSys, the so-called embedding layers account for the majority of memory usage so current systems employ a hybrid CPU-GPU design to have the large CPU memory store the memory hungry embedding layers. Unfortunately, training embeddings involve several memory bandwidth intensive operations which is at odds with the slow CPU memory, causing performance overheads. Prior work proposed to cache frequently accessed embeddings inside GPU memory as means to filter down the embedding layer traffic to CPU memory, but this paper observes several limitations with such cache design. In this work, we present a fundamentally different approach in designing embedding caches for RecSys. Our proposed ScratchPipe architecture utilizes unique properties of RecSys training to develop an embedding cache that not only sees the past but also the “future” cache accesses. ScratchPipe exploits such property to guarantee that the active working set of embedding layers can “always” be captured inside our proposed cache design, enabling embedding layer training to be conducted at GPU memory speed.

CCS CONCEPTS
• Computer systems organization → Neural networks: Heterogeneous (hybrid) systems; Pipeline computing; Parallel architectures; • Computing methodologies → Machine learning; Artificial intelligence.

KEYWORDS
Systems for machine learning, memory architecture, neural network, graphics processing unit (GPU), recommendation system

1 INTRODUCTION
Personalized recommendation systems (RecSys) are one of the most successfully commercialized machine learning (ML) applications in the industry, widely being deployed for online services (e.g., video recommendations from YouTube/Netlix, product recommendations from Amazon). Recent literature observes that RecSys also follows the “larger the model, the better the accuracy” principle, a study from Facebook claiming their production RecSys have increased by tenfold within the past three years [37, 40]. Such scaled-up “up” model development trends rendered state-of-the-art RecSys to reach several hundreds of GBs to TBs of model size [40, 60, 62].

Unlike ML systems for computer vision or natural language processing where GPUs have been the preferred architecture of choice for training, RecSys model’s unique memory requirements make CPUs play a critical role in training these models. In RecSys, the so-called embedding tables account for the majority of memory usage, which is a large lookup table that stores millions to billions of embeddings [37]. High-end GPUs [47] or TPUs [12] tailored for ML training typically employ high-bandwidth but low-capacity memory solutions like HBM [20], whose (only) several tens of GBs of storage falls short in capturing the enormous working set of embedding tables. Consequently, system architectures for training RecSys typically employ a hybrid CPU-GPU system. At a high-level, RecSys models consist of two parts, the frontend embedding layers and the backend DNN layers. Under the CPU-GPU system, the CPU stores the embedding tables inside capacity-optimized CPU memory, thus handling the training process of frontend embedding layers. The GPU then handles the training process of the backend DNN layers using its high-bandwidth (but capacity-limited) memory.

A key limitation of training embedding layers over the CPU memory is that the majority of its execution time is spent conducting a highly memory bandwidth limited operation, which is at odds with the low-throughput CPU memory. An embedding layer conducts the following key primitives during forward propagation (and backpropagation), an embedding gather operation from the embedding tables (and gradient scatter operation to the table) followed by a reduction operation among the gathered embeddings, all of which are highly memory intensive. Consequently, training embedding layers over the CPU-GPU system is known to incur performance bottlenecks [30].

To address these challenges, recent work proposed to leverage locality inherent in the memory accesses to/from embedding tables [24, 25, 34, 61], seeking to alleviate embedding layer’s memory throughput requirement. Prior work observes that embedding table accesses follow a power-law distribution, i.e., a small subset of “hot” embeddings capture a significant portion of the overall accesses to the embedding table. Based on such insight, a CPU-GPU system augmented with a software-managed GPU embedding cache can store the most frequently accessed hot embeddings in fast GPU memory,
which helps reduce the memory bandwidth bottlenecks of CPU-side embedding layer training. As we detail in Section 3.2, however, a CPU-GPU system augmented with a GPU embedding cache still suffers from non-trivial amount of embedding cache misses. Under such circumstances, the CPU-side embedding tables must inevitably be accessed to retrieve the missed embeddings, the latency of which sits in the critical path of the end-to-end training, hurting performance.

In this work, we present a fundamentally different approach in designing software embedding caches for RecSys training. Conventional caches typically employ a history based, speculative cache insertion/replacement policy. This is because it is generally impossible to know what memory accesses will be observed in the future, so caches are generally designed to utilize past cache access patterns to make educated guesses on what may happen moving forward. Under the context of training RecSys, we make the key observation that it is actually possible to "precisely" know when and how many memory accesses to the embedding tables will occur in the future. This is because the training dataset records exactly which indices to utilize for embedding gathers (and gradient scatters) as these will be the primary target for model updates during training, not just for the current but also for all upcoming training iterations. Our proposal utilizes such insight to identify what embedding table accesses will occur in future training iterations and utilize that information to develop an optimal GPU embedding cache that "always hits".

To this end, we propose ScratchPipe, a software runtime system that manages high-bandwidth GPU DRAM as a fast "scratchpad". Our software runtime is carefully designed to systematically bring in the required embeddings from CPU to our GPU scratchpad right before the embedding training procedure is initiated such that embedding table accesses always become hits. ScratchPipe first fetches soon-to-be-accessed embedding table lookup IDs from the training dataset, completely transparent to both the programmer as well as the ML framework (e.g., PyTorch). The fetched lookup IDs are then utilized by the ScratchPipe controller to identify the set of embeddings to proactively prefetch from CPU memory and subsequently copied over to GPU scratchpad. To hide the latency overhead of CPU→GPU embedding prefetching, ScratchPipe collects multiple mini-batches from the training dataset, concurrently processing different stages of multiple training iterations via "pipelined" execution. Because ScratchPipe guarantees the prefetched embeddings are filled into the GPU scratchpad before the on-demand cache accesses are initiated, our proposal provides the illusion of a "GPU-only" system (designed over the baseline CPU-GPU) that enables embedding table accesses to occur at GPU memory speed. As such, ScratchPipe can achieve commensurate performance to a multi-GPU based, model parallel GPU-only system using only a single GPU machine, significantly reducing RecSys training cost.

Overall, ScratchPipe achieves an average $5.1 \times$ (max $6.6 \times$) and $2.8 \times$ (max $4.2 \times$) speedup vs. the baseline hybrid CPU-GPU without and with software-managed GPU embedding caches, respectively. Furthermore, while ScratchPipe only utilizes a single GPU machine for training, we achieve comparable performance to an 8 multi-GPU system that can store all the embedding tables entirely within GPU's high-bandwidth memory (i.e., GPU-only), thus providing a significant reduction in per epoch training cost (avg $4.0 \times$, max $5.7 \times$), enabling cost-effective RecSys training in cloud datacenters.

2 BACKGROUND AND RELATED WORK

2.1 RecSys Models and Embedding Layers

Figure 1 shows a DNN based RecSys, which is composed of two major components: the embedding layer and the DNN layer using multi-layer perceptrons (MLP). RecSys is typically formulated as a problem of predicting the probability of a certain event (e.g., the likelihood of an e-commerce shopper to purchase the recommended product), so the accuracy of RecSys depends on how the unique properties of various input features are captured. An embedding is a projection of a discrete-valued, categorical feature into a vector of continuous real-valued numbers. For instance, different product items sold in e-commerce services are projected from a discrete ID (i.e., each product item is assigned a unique ID for differentiation) into a continuous, real-valued vector representation. Such process is done by training the embeddings using embedding layers. Because the number of unique items falling under a particular feature typically amounts to several millions to billions (i.e., the number scales proportional to the number of products sold in e-commerce or videos available in online video streaming services), an embedding table which stores all embeddings amounts to several tens of GBs of capacity. Because there can be multiple categorical features (e.g., user ID, item ID, ... ) that are helpful in capturing semantic representations, several tens of embedding tables can be utilized per RecSys, rendering the overall model to consume several hundreds of GBs to even TBs of memory capacity [60, 62].

2.2 Training Pipeline for RecSys

Forward propagation. Figure 2(a) shows the key computations conducted during forward propagation of training the embedding layers in RecSys. Using the embedding tables, a discrete, categorical feature is mapped into its corresponding vector representation via embedding gather operations, i.e., utilize a group of sparse feature IDs to index the embedding table and read out the corresponding embeddings. The group of IDs used for gathering embeddings do
not necessarily point to contiguous rows within the embedding table, so an embedding gather operation exhibits a highly sparse and irregular memory access pattern, exhibiting a memory bandwidth limited property. The gathered embeddings are combined with each other via element-wise operations, hence reduced down to a single vector. The embedding reduction is done per each table and the reduced embeddings are combined with the output of bottom MLP layers (whose role is to transform continuous input features into an intermediate feature vector) via the feature interaction stage. The output of feature interaction is handled by the top MLP layer and subsequently processed using softmax functions to predict the final click through rate (CTR), e.g., the probability of an end-user to click the recommended item. The backpropagation stage of MLP layers then derives a set of gradient vectors that are routed back to the embedding layers. Note that the number of gradients to backpropagate is identical to the number of reduced embeddings during forward propagation (e.g., two in Figure 2(a), $G[0], G[1]$).

**Backpropagation.** An interesting property of RecSys training is that the embeddings stored inside embedding tables are the ones subject for model updates, i.e., embedding tables are both read and written during the course of training. Specifically, during a single iteration of forward and backpropagation, it is those gathered embeddings during forward propagation that will be the target of training during backpropagation. This process is illustrated in Figure 2(b), where the two gradients, backpropagated from the backend MLP layers, are used to update multiple rows within the embedding table (i.e., those looked up locations that have been gathered during forward propagation) using a gradient scatter operation. For instance, in Figure 2, the embedding table’s rows 0 and 4 for the first and second batch and rows 0, 2, and 5 for the second batch were gathered during forward propagation (a), so these locations are targeted for gradient scatter update during backpropagation (b). Because any given embedding can be read out multiple times across different batches (e.g., $E[0]$ at row 0 is looked up twice for the first and second batch), multiple backpropagated gradients must be accounted for when deriving the final gradients to use for model updates. This is handled by a series of gradient duplication and coalescing, which is also a memory bandwidth limited operation.

### 2.3 System Architectures for Training RecSys

State-of-the-art RecSys employ large embedding tables which can require up to several TBs of memory (Section 2.1). Since training is a throughput-bound operation, high-end GPUs or TPUs employ bandwidth-optimized but capacity-limited memory solutions like 3D stacked DRAMs (e.g., HBM [20]). As these memory solutions only come with several tens of GBs of storage, it becomes very challenging to store the enormous embedding tables within GPU local memory. Consequently, system architectures for RecSys training typically adopt a hybrid CPU-GPU system (e.g., Facebook’s Zion system [10], Baidu’s AIBox [62]). In such system design point, the capacity-optimized CPU DIMMs are used to store the embedding tables so that the CPU handles the training of memory-intensive embedding layers, whereas the GPU conducts the training of compute-intensive DNN layers. Since the key compute primitives of both forward and backpropagation of embedding layers are highly memory bandwidth limited (Section 2.2), executing them over the slow CPU memory causes significant slowdown in RecSys training.

### 2.4 Related Work

Unlike prior literature focusing on the acceleration of compute-intensive DNN algorithms [1, 4–6, 16, 22, 23, 27, 28, 36, 48], RecSys models pose a unique challenge for computer architects as they incur unprecedented levels of memory capacity and bandwidth demands [15]. TensorDIMM, RecNMP, Centaur, Fafnir, and TRiM [3, 18, 25, 26, 29, 49] propose a near-in-memory processing solution specialized for embedding layer’s gather-and-reduce to overcome the memory bandwidth limitations of this primitive under RecSys inference. More recently, RecSSD explored the efficacy of employing SSDs to cost-effectively store TB-scale embedding tables [57]. To close the wide performance gap between DRAM and SSD, RecSSD employs an in-storage processing unit for embedding gather-and-reduce. Under the context of training, TensorCasting [30] proposed a near-memory processing architecture for RecSys’s embedding layer training.

Because near-in-memory or in-storage processing requires modification to the current hardware/software stack, recent studies explored the possibility of leveraging locality inherent in the read/write accesses from/to embedding tables to cost-effectively reduce embedding layer’s memory bandwidth demands [24, 25, 34, 57, 59, 61].
RecNMP and RecSSD both observe that a small subset of embedding table entries account for a significant fraction of embedding table lookups, exhibiting a power-law distribution. Therefore, these studies employ a lightweight near-memory/in-storage embedding cache that helps filter out the embedding read traffic for the “hot” embedding table entries. MERCIA [34] and SPACE [24] similarly observe locality within the gather and reduction operations of RecSys, proposing memorization [34] and a heterogeneous memory hierarchy [24] respectively for fast inference. There is also recent work that co-designs the RecSys training algorithm with the underlying hardware/software system, reducing memory capacity [61] or memory bandwidth demands [59] of RecSys training. For instance, Yin et al. [61] suggests matrix decomposition and approximation work that co-designs the RecSys training algorithm with the under-backpropagation of embedding layers, the embeddings corresponding to hit IDs (missed IDs) are updated in EMB feature IDs are first copied over to the GPU to evaluate hit/missed IDs. The missed IDs are then copied back to CPU memory for embedding table lookups. Later during the backpropagation of embedding layers, the embeddings corresponding to hit IDs (missed IDs) are updated in EMB, over GPU memory in EMB, over CPU memory. Gray (black) stages are those executed on the GPU (CPU). Red arrows designate CPU→GPU copies.

Figure 4: RecSys training pipeline under (a) baseline CPU-GPU without caching [30], and (b) CPU-GPU augmented with a software-managed GPU embedding cache (EMB$_{gpu}$) that statically stores top-$N$ most-frequently-accessed embeddings of the original embedding table (EMB$_{cpu}$) stored in CPU memory [61]. In (b), the sparse feature IDs are first copied over to the GPU to evaluate hit/missed IDs. The missed IDs are then copied back to CPU memory for embedding table lookups. Later during the backpropagation of embedding layers, the embeddings corresponding to hit IDs (missed IDs) are updated in EMB$_{cpu}$ over GPU memory (in EMB$_{cpu}$ over CPU memory). Complex data hazards so ScratchPipe employs a novel hazard resolution mechanism to guarantee that the algorithmic nature of RecSys training is not altered. While not specifically focusing on recommendation models, there is a rich set of prior literature exploring heterogeneous memory systems for training large-scale ML algorithms [7, 17, 21, 31–33, 50, 53–56]. In general, the key contribution of our ScratchPipe is orthogonal to these prior studies.

3 MOTIVATION

3.1 Locality in Embedding Layer Training

In Figure 3, we show the (sorted) access count of embedding table entries in various real-world RecSys datasets. The embedding table accesses generally exhibit a power-law distribution where a small subset of table entries receive very high access frequency while the remaining entries receive only a small number of accesses. Such long-tail phenomenon is not surprising as it is typically the case where a large fraction of RecSys users are interested in a small portion of the most popular items (e.g., popular products in Amazon or most watched movies in Netflix). While the magnitude of locality is highly dependent on where the RecSys model is being deployed for (e.g., in Criteo Ad Labs, 2% of the embeddings account for more than 80% of all accesses whereas for Alibaba User dataset, 2% of embeddings “only” account for 8.5% of traffic), we generally observe that RecSys datasets indeed exhibit the power-law distribution with a very long-tail.

Unfortunately, the baseline hybrid CPU-GPU is not designed to leverage the unique locality properties of embedding layers [30]. Figure 4(a) shows the key stages of forward and backpropagation under baseline CPU-GPU. As depicted, the memory bandwidth limited embedding gather and gradient scatter operations are all conducted over the slow CPU memory, so the end-to-end training time is primarily bottlenecked on the CPU-side training of embedding layers, consuming significant portion of training time (Figure 5).
We make several important observations from this experiment. First, we assume the GPU embedding cache is large enough to fully cache the entire embedding tables within GPU’s local high-bandwidth memory.

3.2 Software-Managed Embedding Caches

Given the power-law distribution of embedding table accesses, an effective mechanism to exploit such locality is to incorporate a small GPU embedding cache (managed in fast GPU DRAM) that can help reduce the embedding gather/gradient scatter traffic to CPU memory. Figure 4(b) provides an overview of RecSys training under a CPU-GPU system enhanced with a software-managed GPU embedding cache. The key data structures of such software-level cache (e.g., tag, data, other metadata) as well as its cache controller module are all implemented and managed in software over GPU memory using CUDA [44]. In terms of cache management policies, we assume the static cache architecture suggested by Yin et al. [61] where the most-frequently-accessed embeddings (i.e., the top-N leftmost table entries in Figure 3) are chosen for caching in GPU DRAM without getting evicted throughout the entire RecSys training process. Figure 5 shows the normalized end-to-end training time broken down based on where the key stages of training is executed. We make several important observations from this experiment. First, the static GPU embedding cache noticeably reduces the time spent in the memory bandwidth limited embedding layer training. This is because of the hot embeddings hitting in the GPU embedding cache, which helps filter out the number of embedding table accesses serviced by the CPU memory. Unfortunately, there is still a non-negligible amount of time spent in servicing the (embedding cache missed) embedding gather and gradient scatter operations over CPU memory, averaging at 77% to 94% of training time. In particular, the backpropagation of embedding layers corresponding to the missed IDs (the rightmost two black stages in Figure 4(b)) invoke memory bandwidth limited gradient duplication/coalescing/scatters over the slow CPU memory, causing serious latency overheads. On average, the static GPU embedding cache suffers from 12% (high locality dataset) to 91% (low locality dataset) cache miss rates, all of which must be serviced from CPU embedding tables.

One might argue that a larger GPU embedding cache could potentially absorb the majority of traffic to CPU memory and enable the performance of CPU-GPU to reach that of a “GPU-only” system (i.e., the high-bandwidth GPU memory stores all embedding tables, allowing embedding layers to be trained at GPU memory speed). Figure 6 plots the improvements in GPU embedding cache hit rate as we cache a larger fraction of CPU embedding tables. For datasets like Criteo, a small fraction of hot embedding table entries exhibit extremely high locality (Figure 6(d)). Consequently, having larger embedding caches provide only incremental improvements in further absorbing CPU memory traffic. Conversely, for datasets with low locality (Figure 6(a)), achieving >90% cache hit rates would require more than 65% of the embedding table entries to be cached in GPU memory. Given state-of-the-art RecSys require TB-scale memory capacity, caching such high proportion of embedding tables is an impossible design point to pursue given the “meager” tens of GBs of GPU memory.

3.3 Our Goal: Training Embedding Layers at GPU Memory Speed

Our characterization root-caused the memory bandwidth limited embedding layer training as a key limiter in RecSys training’s performance. Caching most-frequently-accessed, hot embeddings inside high-bandwidth GPU memory could alleviate the performance penalties of sparse embedding gathers and gradient scatter operations, but the limited GPU memory capacity makes it impossible to fully capture the active working set of embedding tables. This makes GPU embedding caches to still suffer from the latency overhead of bringing in the missed embeddings from CPU memory, which sits on the critical path of RecSys training and hurting performance. A key objective of our study is to develop a GPU embedding cache that is able to intelligently store (and evict) not just the current but also future embedding table accesses such that its active working set is “always” available within GPU memory by the time embedding layer training is initiated. This would allow the training of embedding layers to occur at the speed of high-throughput GPU memory, drastically improving the performance of end-to-end RecSys training.

4 SCRATCHPIPE ARCHITECTURE

In Section 4.1, we first elaborate on the design objectives behind the proposed ScratchPipe architecture. Section 4.2 then discusses our naive, straw-man architecture which we utilize to point out the important research challenges this paper innovates in. Finally, Section 4.3 presents our ScratchPipe architecture with its implementation details discussed in Section 4.4.
4.1 Design Overview

Design objectives. ScratchPipe is a software runtime system that manages high-bandwidth GPU memory as a fast scratchpad for servicing embedding gather and gradient scatters. Our GPU scratchpad is designed to carefully (and systematically) bring in the required embeddings from CPU→GPU right before the embedding training procedure is initiated, so it functions as an embedding cache that always hits. This allows ScratchPipe to train embedding layers virtually at GPU memory speed, which is a level of performance only achievable when scaling out to multi-GPU systems. In other words, to train embedding layers at GPU memory speed under conventional systems, all the embedding tables must be partitioned and distributed across multiple GPU’s local memory so that embedding table accesses are always captured within the multi-GPUs’ memory pool. A key design objective of ScratchPipe is to reach commensurate performance of such “GPU-only” system using a single GPU machine, thereby significantly reducing RecSys training cost. Later in Section 6.6, we evaluate the advantages of our ScratchPipe vs. multi-GPU systems in terms of training cost reduction.

Key observations. Conventional caches are not able to achieve the aforementioned research objective (i.e., a cache that always hits) because the cache insertion/replacement policy is a reactive one based on a best-effort speculation of what may happen in future memory accesses based on past history. The “optimal” cache design we pursue must know exactly when and how many data elements will be accessed soon and utilize that information to proactively “prefetch” the required data into the cache just before the on-demand cache accesses occur, enabling them to all become cache hits.

Our key observation is that, with respect to embedding layer training, it is actually possible to precisely know when and how many embedding table accesses will occur in the future. More concretely, the sparse feature IDs used for embedding gather and gradient scatters (Figure 2) are already recorded as part of the training dataset. This is because the embeddings corresponding to the sparse feature IDs will be the primary target for model updates, so the training dataset contains the indexing information to the embedding tables, i.e., what memory locations to reference for embedding gather and gradient scatters. In other words, the training dataset is, by design, implemented to provide exactly which rows within the embedding table to read (write) from (to), not just for the current but also for future training iterations.

The novelty of ScratchPipe is the utilization of such information to develop our GPU embedding cache architecture that can “always” fully service embedding gather and gradient scatters over GPU memory. Figure 7 illustrates our key approach in designing ScratchPipe. Conventional GPU embedding caches frequently invoke cache accesses because the cache insertion/replacement policy is a reactive one based on a best-effort speculation of what may happen in future memory accesses based on past history. The “optimal” cache design we pursue must know exactly when and how many data elements will be accessed soon and utilize that information to proactively “prefetch” the required data into the cache just before the on-demand cache accesses occur, enabling them to all become cache hits.

Figure 8: Key steps undertaken during straw-man architecture’s single iteration of RecSys training: The black circles (1-4) refer to the RAW dependencies at the GPU embedding cache (1,2,3) and the CPU embedding table (4). Concurrently processing different stages of multiple training iterations via “pipelined” execution. As depicted in Figure 7, hiding the latency of copying prefetched embeddings from CPU→GPU enables a single RecSys training iteration to be completed every single pipeline “cycle” (the red-colored stages in Figure 7). A key challenge of such pipelined design is that the concurrently executing mini-batches’ embedding table accesses can potentially interfere with each other’s GPU embedding cache lookups, hindering the correct execution of RecSys training algorithm. Consequently, the research problem lies in how to intelligently manage the GPU embedding cache when multiple input mini-batches are concurrently in-flight, without violating the correctness of program execution. In the next subsection, we present a naive, straw-man architecture for ScratchPipe which we utilize as a driving example to discuss its flaws and limitations for realizing the vision behind Figure 7, motivating the important research challenges we address in this work.

4.2 Straw-man Architecture of ScratchPipe

The need for dynamic (not static) embedding caches. The static embedding cache we assume in Figure 4(b) is always filled in with the top-N hot embeddings without eviction. ScratchPipe requires the ability to dynamically determine what embedding table accesses will occur for the current and upcoming mini-batches and utilize that information to 1) proactively prefetch the embeddings (currently missing in the GPU embedding cache) from CPU memory and 2) transfer them to the GPU embedding cache in advance, before the on-demand embedding fetches occur. Designing such dynamic embedding cache requires the following capabilities: 1) determine which among the sparse feature IDs are hits or misses, 2) utilize that information to fetch the missed embeddings from the CPU embedding tables, and 3) choose a number of embeddings (equivalent to the number of missed IDs) to evict from the GPU embedding cache, so that the missed embeddings fetched from the CPU can be filled into the GPU embedding cache. As shown in Figure 8, our straw-man with a dynamic GPU embedding cache goes through the following steps for training:

1. [Query] stage: copies the sparse feature IDs for the current training iteration from CPU→GPU and query the GPU embedding cache, determining the hit/miss IDs.
2. [Collect] stage: utilizes the missed IDs to fetch the corresponding embeddings from CPU embedding tables. Concurrently, the GPU fetches the same number of victim embeddings from the GPU embedding cache so that 1) the evicted embeddings can be written backed into CPU’s embedding tables, and 2) the missed embeddings fetched from the CPU can be inserted into the evicted entries. The reason why the
GPU embedding cache must writeback the evicted embeddings to the CPU memory because the GPU embedding cache holds dirty copies of the trained embeddings (i.e., the trained embeddings).

(3) [Exchange] stage: copies the GPU cache missed embeddings from CPU→GPU while also simultaneously copying the evicted embeddings from GPU→CPU over PCIe.

(4) [Insert] stage: uses the CPU→GPU exchanged embeddings to update the CPU embedding tables with the (GPU embedding cache) evicted embeddings while the GPU fills in the missed embeddings into the GPU embedding cache.

Once we reach the [Insert] stage, the GPU embedding cache now holds the complete set of embeddings required to go through forward and backpropagation of embedding layers. Specifically, the [Embedding Forward] stage gathers all the required embeddings from the GPU embedding cache (which will all be hits) and goes through the remaining MLP forward and backpropagation. Once the final gradients are ready, the [Parameter Update] stage overwrites the corresponding rows in the GPU embedding cache with the updated model values. Consequently, all embeddings inserted into the (dynamic) GPU embedding cache are subject for training, thus any entries that get evicted from the embedding cache must be written back into the main CPU embedding tables.

Limitations of straw-man architecture. A critical challenge of straw-man is that the [Cache Management] stages sit in the critical path of training. As discussed in Figure 7, our goal is to collect multiple training mini-batches and concurrently process different stages of multiple training iterations via pipelined execution, which will enable the embeddings collected from [Query→Collect] stages to have the effect of being prefetched into the GPU. Unfortunately, the multiple stages in straw-man invoke several read-after-write (RAW) data dependencies at both the GPU embedding cache and the CPU embedding tables, which could prevent straw-man from concurrently executing different stages in Figure 8. Specifically, the GPU embedding cache becomes a source of RAW dependencies during the following pair of stages:

- Parameter Update (W) → [Embedding Forward] (R) (RAW-1 in Figure 8): write trained embedding values to update the embedding cache → read embeddings from the embedding cache for forward propagation.
- Parameter Update (W) → [Collect] (R) (RAW-2): write trained embedding values to update the embedding cache → read victim embeddings from embedding cache.
- Insert (W) → [Collect] (R) (RAW-3): write missed embeddings into embedding cache → read out victim embeddings from embedding cache.

Similarly, the CPU embedding table becomes another source of RAW dependency during [Insert] (W) (i.e., write evicted embeddings into CPU embedding table) and [Collect] (R) (i.e., read missed embeddings from CPU embedding table), RAW-4 in Figure 8. All of these RAW dependencies are naturally resolved without any complications when different training iterations are sequentially executed. When we seek to pipeline the straw-man architecture however, the RAW dependencies cause a data hazard inside the pipeline and prevents the correct execution of RecSys training (Figure 9).

4.3 “Pipelined” ScratchPipe Architecture

Our final proposition for ScratchPipe, its “pipelined” version, holistically addresses all the shortcomings of our straw-man with our principled design approach. For clarity of explanation, we henceforth refer to the pair of stages that incur RAW dependencies (1 to 4) as RAW dependent stages.

The pipelined ScratchPipe incorporates the [Plan] stage (as a substitute for [Query] in straw-man) in its 6-stage pipeline (Figure 10), which provides the following functionality:

(1) The [Plan] stage is the main control unit that plans out in advance which embeddings to gather and fill (from CPU embedding tables to GPU embedding cache) and evict and write-back (from GPU embedding cache to CPU embedding tables) during the remaining pipeline stages, specifically during [Collect] and [Insert].

(2) All the procedures undertaken during the straw-man’s [Query] stage are also conducted in [Plan], i.e., copying the sparse feature IDs for the current mini-batch from CPU→GPU and determining the hit/missed IDs by querying the GPU embedding cache.

Overall, the goal of ScratchPipe’s [Plan] stage is to remove any RAW hazards from occurring inside the pipeline so that ScratchPipe’s prefetching latency is effectively hidden without causing any complications to RecSys training.

RAW dependencies in “sparse” embedding tables. Recall that a unique property of embedding table accesses is that they are extremely sparse, i.e., a single training iteration only touches upon several tens of thousands of rows within the several millions to billions of entries in the embedding tables. The implication of such property from a RAW data dependency perspective is as follows: as long as the row IDs used for the reads/writes to/from the GPU embedding cache and the CPU embedding tables do not overlap across RAW dependent stages, we are able to completely remove the effective data dependencies, thus resolving any data hazards from occurring. The key is to make sure that the set of row IDs used for "read" operations (R) from GPU embedding cache or CPU embedding table do not match the row IDs used for "write" operations (W) across RAW dependent stages. Below we elaborate on
Removing RA\textsuperscript{-}2 / \textsuperscript{-}3. To prevent RA\textsuperscript{-}2 / \textsuperscript{-}3 from causing data hazards, ScratchPipe should guarantee that the GPU embedding cache entries scheduled for updates through writes (i.e., updated during [Parameter Update] or [Insert] (W)) are never prematurely chosen as victims to be read out from the embedding cache and written back into CPU embedding tables. ScratchPipe handles both RA\textsuperscript{-}2 / \textsuperscript{-}3 dependencies through the following mechanism: when [Plan] chooses victims to evict out of the GPU embedding cache, ScratchPipe does not consider those set of input sparse feature IDs used during the previous three training iterations (i.e., the distance between the [Training] stage and the [Collect] stage) as victims. At steady-state, a total of six training mini-batches (corresponding to six sets of input sparse feature IDs) are concurrently being processed – but executing at different stages. When an input mini-batch enters the [Plan] stage, the ScratchPipe controller examines three previous mini-batches (executing in [Collect-Exchange-Insert] stages from the [Plan] stage’s perspective) and rules out all of their input sparse feature IDs from embedding cache eviction candidates. This allows the input mini-batch currently executing in [Plan] to never evict (i.e., read (R)) any one of the(embeddings that will be updated ((W)) by previous input mini-batches executing in RAW dependent stages, effectively eliminating hazards.

Removing RA\textsuperscript{-}4. Preventing hazards for RA\textsuperscript{-}4 is a matter of controlling the “read” part of the RAW dependency (i.e., “writes” to GPU embedding caches are done by previous input mini-batches which the [Plan] stage has no control over, so ScratchPipe controller judiciously rules out potentially hazard invoking row IDs from eviction to prevent RA\textsuperscript{-}4. Removing RA\textsuperscript{-}4 is the opposite as the source of RAW dependency is at the CPU embedding tables. Here “writes” to the CPU embedding tables is an artifact of the [Plan] stage choosing GPU embedding cache victims. Because the evicted entry’s write-back to embedding tables occurs during the [Insert] stage, preventing RAW hazards for RAW\textsuperscript{-}4 is a matter of controlling the “write” part of the RAW dependency and making sure controlling the RAW dependency and making sure upcoming, future input mini-batches do not conflict with the currently chosen eviction candidates (e.g., from the perspective of the mini-batch executing in [Insert], the mini-batch executing in the RAW dependent [Collect] stage is a future input). ScratchPipe removes RAW\textsuperscript{-}4 via the following mechanism: when [Plan] chooses victims to evict out of the GPU embedding cache, ScratchPipe does not consider those set of input sparse feature IDs used during the next two upcoming training iterations (i.e., the distance between [Insert] and [Collect]) as victims.

Putting everything together. As depicted in Figure 10, ScratchPipe systematically evaluates potential RAW hazards and removes them via our sliding window based [Plan] stage. When an input mini-batch enters the [Plan] stage, the three previous (past-window) as well as the next two (future-window) mini-batches of sparse feature IDs are examined. By creating a superset of the sparse feature IDs falling under the past-/future windows, the [Plans] control unit rules out the IDs included in the superset from cache eviction candidates, preventing RAW hazards from occurring in subsequent stages.

### 4.4 Implementation

We now discuss ScratchPipe’s implementation details using Figure 11 as an illustrative example. Algorithm 1 provides a pseudocode of the key operations conducted by ScratchPipe’s controller over its key data structures. For brevity and consistency with Figure 11, Algorithm 1 assumes the Hold mask only holds three mini-batches falling under the past-window.

**GPU scratchpad design.** As depicted, ScratchPipe’s GPU embedding cache (i.e., the GPU scratchpad) is implemented using 1) a data array to store the cached embedding vectors (denoted Storage), and 2) a (key, value) store called Hit-Map which returns the cache query results as a hit or a miss, i.e., the (key, value) stores the cached embedding’s original sparse feature ID (key) and the index to locate the cached embedding within the Storage array (value). Whenever any given mini-batch enters the [Plan] stage, the GPU scratchpad’s Hit-Map is queried to derive the hits/misses, the result of which is used to schedule which embeddings to gather from the CPU embedding table (if any) and which entries to evict from the GPU scratchpad (if any).

A unique property of our GPU scratchpad design is that the status of the Hit-Map and Storage is updated in a (purposefully) asynchronous and delayed manner. As depicted in Figure 11, the status of the Hit-Map is updated whenever a new mini-batch is processed at [Plan], whereas the Storage array gets updated when a mini-batch enters [Insert] with a Hit-Map miss. This is an artifact of ScratchPipe’s pipelined design, rendering the status of Hit-Map to always reflect the embedding caching status of the Storage array “four” cycles later in the future (i.e., the distance between [Train] and [Plan]). For instance, the second mini-batch of ID 3010/7089 is returned as miss/hit when querying the Hit-Map during the 2nd cycle, even though the Storage array is still left vacant (Figure 11(b)). Such discrepancy between the status of Hit-Map and Storage is intentional because 1) the Storage array should not have
yet to be updated with the first mini-batch’s query of ID 7089/2021 as it has not arrived at the [Insert] stage, and 2) the second mini-batch should still be able to see the precise caching status of GPU scratchpad (i.e., the status assuming the first mini-batch completed its training) so that it can accurately determine the set of embeddings to prefetch from CPU, even though it is currently at [Plan] stage.

**Hold masks for removing RAW hazards.** ScratchPipe removes RAW hazards by having the [Plan] stage examine the three previous (past-window) and two next (future-window) mini-batches’ embedding table look-up IDs falling under the current sliding window (centered around [Plan]), making sure those cached locations are not evicted from the GPU scratchpad (Section 4.3). As shown in Figure 11, we employ a data structure named **Hold mask**, which is a bitmask (number of bits within the bitmask is equal to the number of cacheable slots in Storage array) the [Plan] stage utilizes to designate locations within the scratchpad’s Storage array that the current mini-batch will be utilizing at the [Train] stage. The Hold mask is designed using a circular queue to accommodate its sliding window based operation, storing a set of six bitmasks that keep track of the past three, one current, and two future mini-batches’ cached locations in Storage – ones that should not be targeted for eviction while the sliding window is effective. When the control unit in [Plan] stage needs to select a victim for GPU scratchpad eviction, the Hold mask is examined. The victim candidates are chosen as the Storage array locations that correspond to the Hold mask’s bit-locations set with a value of ‘0’. This means none of the sparse feature IDs that fall under the current sliding window asked to **hold** this location inside the GPU scratchpad, hence is allowed to be evicted. For instance, E[2021] stored as the 3rd element inside the Storage array is targeted for eviction at the 5th cycle as the corresponding location in the Hold mask is “000” after the 4th cycle (Figure 11(d,e)).

It is worth pointing out that, in order for our ScratchPipe architecture to guarantee that the bit-locations within the Hold mask set to non-zero values are never targeted for scratchpad eviction, the Storage array should be large enough to accommodate the worst-case GPU scratchpad usage within the processing of six mini-batches falling under the current window. We quantify the implementation overhead of our GPU scratchpad in Section 6.4. Furthermore, Section 6.5 evaluates ScratchPipe sensitivity to model configurations, replacement policies, etc.

---

**Algorithm 1 ScratchPipe Pipeline Controller**

1. ▶ Key data structures for ScratchPipe control
2. HitMap = [(key,value) storage to index cached embeddings in storage]
3. HoldMask = [circular queue tracking cached locations]
4. HoldMaskWidth ← 3
5. ▶ Step A: A new mini-batch enters the [Plan] stage
6. MiniBatch ← PipelineMiniBatchQueue("Plan")
7. ▶ Step B: Advance HoldMask by one cycle
8. for i ← 1 to CacheSize do
9. HoldMask[i] ← HoldMask[i] >> 1
10. end for
11. ▶ Step C: Iterate through all sparse IDs in mini-batch to determine hits and misses
12. for i ← 1 to NumberOfSparseIDsWithinMiniBatch do
13. ▶ If a sparse ID is found in HitMap, set the corresponding HoldMask
14. if HitMap[MiniBatch[i]] is found in HitMap then
15. holdIdx ← HitMap[MiniBatch[i]]
16. HoldMask[holdIdx] → HoldMask[holdIdx] | HoldMaskWidth ← 1
17. ▶ If sparse ID misses, select victim index whose HoldMask is '0'
18. else
19. evictIdx ← CHOOSE_VICTIM(HoldMask)
20. HitMap[MiniBatch[i]] ← evictIdx
21. HoldMask[evictIdx] ← HoldMask[evictIdx] | HoldMaskWidth ← 1
22. end if
23. end for

---

**Figure 11:** Example showing key data structures used in ScratchPipe’s design. The gray-colored Entity IDs stored in the Storage array designates an embedding located at the row ID i, scheduled for a Hit operation to Storage, is propagating through the pipeline at stage S = [Collect, E = [Exchange], I = [Insert]]. For ease of explanation, the Hold mask only shows three bitmasks tracking the three mini-batches falling under the past-window (no future-window). The red (blue) ‘1’s in the Hold mask means a hit (miss) occurred at the Hit-Map.
5 METHODOLOGY

Hardware. ScratchPipe is evaluated on a server containing Intel Xeon E5-2698v4 (256 GB DDR4, 76.8 GB/sec DRAM bandwidth) and NVIDIA’s V100 (32 GB, 900 GB/sec of DRAM bandwidth). The CPU and GPU communicates over PCIe(gen3) with 16 GB/sec of communication bandwidth.

Software. Our software runtime system is designed using a combination of PyTorch (v1.8.0) [51] for modeling the training dataset loader and the embedding layers with NVIDIA’s cuBLAS/cuDNN [45, 46] for modeling DNN layers. Since ScratchPipe is implemented purely in software, we measure end-to-end wall clock time when reporting performance.

Benchmarks. Because real-world traces used for RecSys training are not publicly available, we generate a synthetic embedding table access trace as follows. As discussed in Section 3.1, the locality inherent in RecSys datasets varies significantly depending on its application domain. To properly reflect the diverse locality characteristics of different RecSys models, the sorted access count of embedding table entries in various real-world datasets (Figure 3) are utilized to generate a wide ranging set of probability density functions (PDFs) that quantifies an embedding table entry’s likelihood of lookup. Depending on the dataset, some PDFs exhibit low locality (e.g., User table in Alibaba [2]) while others exhibiting medium, or high locality (e.g., Criteo [9]). To properly capture a wide range of RecSys embedding table access patterns in our evaluation, we use these PDFs to generate three types of embedding table access traces, exhibiting low, medium, and high locality. We additionally add a random trace (i.e., embedding table access ID’s are randomly generated) to compare against low, medium, and high locality traces. These input traces are fed into our baseline RecSys model to generate four distinct benchmarks to stress test the performance of our evaluated cache architectures. The baseline RecSys model configuration is established using DLRM MLPerf [11] as well as representative RecSys models published by prior work [30, 40, 42], which has eight embedding tables, each table containing ten million embedding vector entries (each embedding sized as a 128-dimensional vector), amounting to 40 GB of total model size. The default RecSys model conducts 20 embedding gathers per each table with a batch size of 2048. In Section 6.5, we study the sensitivity of ScratchPipe when deviating from these default configurations.

Embedding cache size. The size of GPU-side embedding cache has paramount effect on overall performance. Since high-end server class GPUs typically come with several tens of GBs, the GPU embedding cache can practically house < 10% of the several hundreds to thousands of GB scale RecSys models. We follow the methodology adopted by prior work (e.g., the static embedding cache proposed by Yin et al. [61] study 0.01–10% caching of CPU-side embedding tables) and study ScratchPipe’s effectiveness across a range of 2 – 10% cache size.

6 EVALUATION

We explore four design points: two baseline architectures, 1) CPU-GPU without caching, 2) a CPU-GPU with static GPU embedding cache, and two design points from our proposal, 3) our straw-man architecture without pipelining, and 4) ScratchPipe with pipelining. Note that ScratchPipe does not change the algorithmic properties of stochastic gradient descent (SGD) training so the total training iterations required to reach a given target CTR accuracy is identical between baseline and ScratchPipe.

6.1 Latency Breakdown

Before discussing end-to-end performance of ScratchPipe, we analyze its efficacy in addressing baseline architecture’s bottlenecks. Figure 12 shows a latency breakdown of our studied workloads. As discussed in Section 3.2, our two baselines spend significant fraction of training time in conducting the memory bound embedding layer training at the low throughput CPU memory. While storing hot entries in GPU’s embedding cache does help alleviate the bottlenecks of CPU-side training (with larger caches generally helping reduce CPU-side training time), it fails to overcome the fundamental limitations of slow CPU memory as any embedding table accesses that miss in the GPU cache must be routed to CPU memory to conduct not just the embedding gathers (forward propagation) but more importantly the highly memory bound gradient duplication/coalescing followed by scatter updates during backpropagation (Figure 2(b), Figure 4(b)). As shown in Figure 12(b), ScratchPipe substantially reduces latency because the memory intensive embedding layer training is always conducted over our high-bandwidth GPU scratchpad. In ScratchPipe, the only time CPU memory is accessed is when the embeddings to be prefetched into GPU scratchpad are [Collect]ed and when the updated models are evicted and [Insert]ed into CPU embedding tables. While these two stages do incur high latency within ScratchPipe’s cache management step, the absolute time spent in interacting with the CPU memory is significantly reduced compared to the baseline static caching, all thanks to our GPU scratchpad.

6.2 Performance

Figure 13 shows ScratchPipe’s end-to-end speedup. Across all the data points we explore, ScratchPipe achieves an average 2.8× (max 4.2×) speedup vs. static caching, with the magnitude of performance improvement gradually reducing as input traces exhibit higher locality. This is expected because the more locality a given dataset has, the more ScratchPipe is able to leverage its pipeline.
Within the sliding window overlap with each other, this amounts to

$$\text{(number of tables} \times \text{embedding vector size} \times \text{number of concurrent mini-batches}) = (8 \times 20 \times 2048 \times 128 \times 4 \times 8) = 960 \text{ MB of Storage space provisioned for holding not-to-be-evicted embeddings. In reality however, the active working set to actually hold is significantly smaller as many of the IDs gathered within the sliding window become hits. In addition, the Hit-Map also incurs an additional < 1 GB of memory space with other miscellaneous data structures accounting for < 300 MB, amounting to an aggregate < 4 GB of additional GPU side memory allocations under ScratchPipe.

6.3 Energy-Efficiency

ScratchPipe is a purely software-based architecture demonstrated over existing hardware/software stack. To measure system-level power consumption, we utilize pcm-power [19] for CPU’s socket-level power consumption and NVIDIA’s nvidia-smi [43] for GPU power consumption. The aggregated power is multiplied with execution time to derive energy consumption. As depicted in Figure 14, ScratchPipe’s significant training time reduction directly translates into energy-efficiency improvements.

6.4 Implementation Overhead

As discussed in Section 4.4, ScratchPipe requires the Storage array within our GPU scratchpad to at least be large enough to accommodate all the embeddings gathered across the six input mini-batches currently falling under the sliding window. Under our default RecSys model configuration, in the worst case where none of the IDs within the sliding window overlap with each other, this amounts to (number of tables \times number of gathes per table \times mini-batch size \times embedding vector size) = (8 \times 20 \times 2048 \times 128 \times 4 \times 8) = 960 \text{ MB of Storage space provisioned for holding not-to-be-evicted embeddings. In reality however, the active working set to actually hold is significantly smaller as many of the IDs gathered within the sliding window become hits. In addition, the Hit-Map also incurs an additional < 1 GB of memory space with other miscellaneous data structures accounting for < 300 MB, amounting to an aggregate < 4 GB of additional GPU side memory allocations under ScratchPipe.

6.5 Sensitivity

This section evaluates ScratchPipe’s robustness to RecSys model configuration or design parameters of ScratchPipe. Embedding vector dimension. Although our default embedding vector dimension is sized at 128, several prior work employs smaller [25] or larger [8, 40] dimension sizes in their RecSys. Figure 15(a) shows ScratchPipe’s speedup when sweeping the embedding dimension size from 64 to 256. In general, the performance benefits of ScratchPipe remains intact across embedding dimension size with larger speedups achieved when the dimension size gets larger. This is because larger embeddings incur an even higher memory bandwidth pressure, rendering baseline to suffer more while ScratchPipe’s scratchpad providing higher performance benefits. Number of embedding table lookups. Figure 15(b) summarizes the speedup ScratchPipe achieves when the number of embedding table gathers are swept from 1 to 50. As the number of lookups are increased to 50, the embedding layer causes an even more severe performance bottlenecks due to its increase in memory traffic. Consequently, ScratchPipe achieves an even higher speedup with an average 3.7 \times (max 5.6 \times). At the other end of the spectrum, we experiment an embedding table lookup size of 1 which makes the RecSys model suffer less from the embedding layers. ScratchPipe still performs better than our two baselines albeit with less improvements in performance. Generally, ScratchPipe is shown to be highly robust across a wide range of embedding table lookup sizes.

Cache replacement policy, MLP-intensive RecSys models, etc. In addition to the aforementioned sensitivity studies, we also tested ScratchPipe’s robustness under various other design points: 1) changing the GPU scratchpad’s replacement policy from our default LRU (least-recently-used) policy to a random eviction or LFU (least-frequently-used) policy, 2) evaluating the effectiveness of ScratchPipe under more MLP-intensive (and less embedding intensive) models, 3) training under larger or smaller batch sizes, etc. In general, we confirm ScratchPipe’s robustness across a wide range of sensitivity studies. We omit the results for brevity.
Table 1: Training cost of ScratchPipe vs. multi-GPU with 8 V100 GPUs.

| Dataset | System | AWS Instance | Price/hr | Iter. Time | 1M Iter. Cost |
|---------|--------|--------------|----------|------------|---------------|
| Random  | ScratchPipe | p3.2xlarge | $3.06 | 14.82 ms | $40.64 |
|         |         | 8 GPU | $3.06 | 16.22 ms | $41.11 |
| Low     | ScratchPipe | p3.2xlarge | $3.06 | 44.70 ms | $37.99 |
|         |         | 8 GPU | $3.06 | 16.22 ms | $37.99 |
| Medium  | ScratchPipe | p3.2xlarge | $3.06 | 29.68 ms | $25.23 |
|         |         | 8 GPU | $3.06 | 17.52 ms | $13.12 |
| High    | ScratchPipe | p3.2xlarge | $3.06 | 26.34 ms | $22.39 |
|         |         | 8 GPU | $3.06 | 18.61 ms | $12.66 |

6.6 Training Cost Reduction vs. Multi-GPUs

We now discuss ScratchPipe’s benefits in reducing RecSys training cost. Training cost estimates are based on AWS EC2 pricing for P3 instances containing NVIDIA V100 GPUs (Table 1). ScratchPipe does not change the training algorithm so the total number of training iterations required to reach the same level of RecSys algorithmic accuracy are identical between ScratchPipe and the 8 GPU system. As such, we evaluate the training costs for the evaluated systems over 1 million training iterations. The 8 multi-GPU system partitions the embedding tables across 8 GPUs’ HBM for model-parallel training of embeddings while the backend MLP layers are trained using data-parallelism, enabling end-to-end “GPU-only” training. Despite using 8 GPUs, however, the multi-GPU system can only reduce 29%/40%/64%/66% of training time vs. ScratchPipe for high/medium/low/random datasets. Consequently, ScratchPipe consistently achieves significant training cost reduction with an average 4.0× (maximum 5.7×) savings, with more cost savings achieved with higher locality.

6.7 Discussion

Applicability of single CPU-GPU ScratchPipe. Recent literature from Facebook elaborate on the scale of their deployed RecSys model size, exhibiting up to 100× difference in number of embedding tables per model [14, 39]. Specifically, RecSys models for content filtering are known to be smaller sized than those for ranking [15], enabling the overall memory footprint to fit within a single-node’s CPU memory capacity. Given content filtering’s wide applicability across various web-based applications (e.g., filter down movies/news/products to recommend to user), we expect our single CPU-GPU ScratchPipe to be widely applicable as-is over a variety of use cases, providing significant reduction in TCO vs. multi-GPUs.

ScratchPipe for multi-GPU training. In Section 6.6, we demonstrated the advantages of employing single GPU ScratchPipe in terms of TCO reduction vs. multi-GPU systems. While this paper focuses on the single GPU based ScratchPipe design point, for the completeness of our study, we nonetheless discuss the practicality of extending our solution for multi-GPU systems below. A popular approach in partitioning and parallelizing RecSys among multi-GPUs is to employ table-wise model-level parallelism (i.e., partition distinct set of embedding tables across different GPUs) and have each GPU locally go through the embedding layer forward/backpropagation as usual, i.e., each partitioned embedding table is locally treated as an independent embedding table from each GPU’s perspective [40]. ScratchPipe is designed to handle GPU cache management in a per embedding table granularity, i.e., RecSys with N embedding tables will have N instance of ScratchPipe’s cache manager module instantiated. As existing model-parallelization for embeddings are carefully designed to enable each GPU to locally handle forward and backpropagation of embedding layers independently with minimum communication, there is no further inter-GPU RAW hazards or any reordering of embedding lookup indices required for correctness and we expect that ScratchPipe can be seamlessly integrated for multi-GPU training. That being said, while extending ScratchPipe for multi-GPU systems is a viable design point to explore, it is likely not going to be cost-effective in terms of TCO reduction. This is because parallelizing the DNNs over multi-GPUs only provides incremental end-to-end performance improvements (i.e., even with a single GPU, the DNNs are not the most crucial bottleneck, Figure 12). Consequently, ScratchPipe over multi-GPUs can severely underutilize the abundant GPU compute throughput, leading to lower training cost savings. A detailed, quantitative evaluation of such is beyond the scope of this paper and we leave it as future work.

7 CONCLUSION

This paper proposes ScratchPipe, our unique GPU scratchpad based pipelined RecSys training system. We first provide a detailed characterization on the locality properties of real-world RecSys datasets, root-causing embedding layer training as a key performance bottleneck. We then present several of our unique observations that motivate our ScratchPipe design, e.g., using the RecSys training dataset to precisely estimate the past, current, and future embedding table access patterns. Our GPU scratchpad memory utilizes such property to proactively prefetch soon-to-be-accessed embeddings into GPU memory, enabling an embedding cache architecture that always hits, drastically reducing the time spent conducting embedding layer training for RecSys. We evaluate our purely software-based ScratchPipe over real systems, achieving an average 2.8× (max 4.2×) speedup vs. prior GPU embedding cache.

ACKNOWLEDGMENTS

This research is partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1A2C2091753), the Engineering Research Center Program through the NRF funded by the Korean government MSIT under grant NRF-2018R1A5A1059921, and by Samsung Advanced Institute of Technology (SAIT). We also appreciate the support from Samsung Electronics Co., Ltd. Minsoo Rhu is the corresponding author.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Civlutil: Ineffectual-neuron-free Deep Convolutional Neural Network Computing. In Proceedings of the International Symposium on Computer Architecture (ISCA).
[2] Alibaba. 2018. User Behavior Data from Taobao for Recommendation. https://tianchi.aliyun.com/dataset/dataDetail?dataId=649.
[3] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Sung-Kyu Lim, Hyesoon Kim, et al. 2021. Falnit: Accelerating Sparse Gathering by Using Efficient Near-memory Intelligent Reduction. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA).
[4] Tianshi Chen, Zidong Du, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).
[5] Junyi Chen, Tao Luo, Shaoli Liu, Shijun Zhang, Liangqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A
Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Iia Cherniavska, Yinghui Lu, RaghuRaman Krishnamoorthi, Anshu Yu, Volodymyr Kondratenko, Stephanie Pereira, Xuanjie Chen, Wenlin Chen, Vijay Raw, Bill Jia, Liang Xiong, and Misho Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. In arxiv.org.

[43] NVIDIA. 2011. NVIDIA System Management Interface (nvidia-smi). https://developer.nvidia.com/nvidia-system-management-interface.

[44] NVIDIA. 2016. NVIDIA CUDA Programming Guide.

[45] NVIDIA. 2019. cuBLAS Library. (2019).

[46] NVIDIA. 2016. NVIDIA CUDA Programming Guide.

[47] NVIDIA. 2019. cuDNN: GPU Accelerated Deep Learning.

[48] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In Proceedings of the International Symposium on Computer Architecture (ISCA).

[49] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In Proceedings of the International Symposium on Microarchitecture (MICRO).

[50] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[51] PyTorch. 2019. http://pytorch.org.

[52] Samyam Rajbhandari, Olutunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZERO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In arxiv.org.

[53] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA).

[54] Minsoo Rhu, Natalia Gimelshein, Jason Clemens, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In Proceedings of the International Symposium on Microarchitecture (MICRO).

[55] Minsoo Rhu, Mike O’Connor, Niladri Chatterjee, Jeff Poel, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA).

[56] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP).

[57] Mark Wilkening, Uditi Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[58] Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In Proceedings of Machine Learning and Systems (MLSys).

[59] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-Precision Embedding Using a Cache. In arxiv.org.

[60] Xiuyang Yi, Yi-Fan Chen, Sukriti Ramesh, Vinu Rajashekhra, Lichan Hong, Noah Fiedel, Nandini Seshadri, Lukasz Heldi, Xiang Wu, and EH Chi. 2018. Factorized Deep Retrieval and Distributed TensorFlow Serving. In Proceedings of Machine Learning and Systems (MLSys).

[61] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. 2021. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. In Proceedings of Machine Learning and Systems (MLSys).

[62] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In Proceedings of the ACM International Conference on Information and Knowledge Management.