PreSto: An In-Storage Data Preprocessing System for Training Recommendation Models

Yunjae Lee[†]
School of Electrical Engineering
KAIST
yunjae408@kaist.ac.kr

Hyeseong Kim[†]
School of Electrical Engineering
KAIST
hyeseong.kim@kaist.ac.kr

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

***Abstract*—Training recommendation systems (RecSys) faces several challenges as it requires the "data preprocessing" stage to preprocess an ample amount of raw data and feed them to the GPU for training in a seamless manner. To sustain high training throughput, state-of-the-art solutions reserve a large fleet of CPU servers for preprocessing which incurs substantial deployment cost and power consumption. Our characterization reveals that prior CPU-centric preprocessing is bottlenecked on feature generation and feature normalization operations as it fails to reap out the abundant inter-/intra-feature parallelism in RecSys preprocessing. PreSto is a storage-centric preprocessing system leveraging In-Storage Processing (ISP), which offloads the bottlenecked preprocessing operations to our ISP units. We show that PreSto outperforms the baseline CPU-centric system with a $9.6\times$ speedup in end-to-end preprocessing time, $4.3\times$ enhancement in cost-efficiency, and $11.3\times$ improvement in energy-efficiency on average for production-scale RecSys preprocessing.**

***Index Terms*—Recommendation system, computational storage device, near data processing, neural network**

I. Introduction

Deep neural network (DNN) based machine learning (ML) algorithms have demonstrated their effectiveness in a wide range of application domains. Among the successfully deployed ML applications, recommendation systems (RecSys) have emerged as a highly effective tool for online content recommendation services. Such rising demand for recommendation services has rendered hyperscalers to dedicate significant resources to the development and training of diverse RecSys models to ensure high-quality inference services. Unlike latency-optimized ML inference, training algorithms are throughput-hungry workloads that favor high-performance, throughput-optimized accelerators like GPUs. However, these power-hungry GPUs account for a large portion of ML system's operating expenses, so maintaining high GPU utilization becomes critical for lowering TCO (total cost of ownership). Unfortunately, keeping the RecSys training pipeline busy with minimal GPU idle time requires the "data preprocessing" stage to preprocess an ample amount of raw data, so that the preprocessed, train-ready tensors can be fed into the GPU in a seamless manner.

Traditionally, the RecSys training pipeline employed *offline* data preprocessing where the raw data retrieved from the storage system is transformed into train-ready tensors in advance and gets archived at a separate storage space for future usage. However, the proliferation of petabyte-scale data and the wide variety of RecSys models developed by ML engineers have rendered offline preprocessing to incur an intractable amount of overhead [70]. Specifically, it becomes increasingly difficult to provision the substantial storage space required to store all the data preprocessed offline, while also adapting to changes in the newly developed RecSys models.

These challenges have triggered a shift towards *online* preprocessing, which involves preprocessing the raw data "on-the-fly". Online preprocessing obviates the need to separately store the preprocessed data, so it helps better respond to changes in the model architecture [70]. Nevertheless, these online preprocessing approaches introduce several system-level challenges, potentially causing a throughput mismatch between data preprocessing and ML model training. Consider a scenario where preprocessing and training jobs are *co-located* within the same GPU training server node, i.e., preprocessing is undertaken on the host CPU that also manages the GPU-side training job. If the CPU does not have a high enough computation power for preprocessing, it fails in generating sufficient amount of train-ready tensors that the GPU can consume, leading to significant GPU underutilization (less than 20% GPU utility, Section III-A). To address these challenges, Zhao et al. [70] and Audibert et al. [5] suggest a *server disaggregation* solution where a pool of CPU servers is reserved for preprocessing purposes. Disaggregating CPU servers for preprocessing allows hundreds to thousands of CPU cores to be allocated *on-demand*, even for a single data preprocessing job, effectively closing the performance gap between preprocessing and model training thereby minimizing GPU idle time [25], [70]. However, this baseline "CPU-centric" disaggregated preprocessing incurs significant deployment cost and power consumption due to the large number of pooled CPU servers.

Given this landscape, an important objective of our work is to characterize baseline CPU-centric disaggregated data preprocessing systems targeting production-scale RecSys models, root-causing its critical system-level challenges. A key observation we make is that the majority of data preprocessing time is spent conducting *feature generation* and *feature normalization* operations, which inherently contain high *inter-/intra-feature parallelism*. However, the latency-optimized CPU architectures, the de facto standard in data preprocessing, fail to fully exploit the abundant inter-/intra-feature parallelism in feature gener-

[†] Co-first authors who contributed equally to this research.

ation and normalization, leading to sub-optimal performance. This in turn leads to the feature generation and normalization to account for 79% of the RecSys data preprocessing time, causing the most significant performance bottleneck. To make up for the meager preprocessing throughput provided with CPUs, the data preprocessing stage necessitates a large number of CPU cores (up to several hundreds) to be allocated so that its aggregate preprocessing throughput matches the throughput demands of GPU's model training stage, which leads to substantial deployment cost.

In this work, we propose to employ *accelerated* computing for RecSys data preprocessing to fundamentally address its system-level challenges at low cost. Because the abundant inter-/intra-feature parallelism in data preprocessing is well-suited for domain-specific acceleration, our first key proposal is to *offload* the time-consuming feature generation and normalization operations to our accelerator for high-performance data preprocessing. An important design decision still remains, however, regarding *where* our data preprocessing accelerator should be placed within the overall system architecture. State-of-the-art data preprocessing solutions employ disaggregated CPU servers to dynamically allocate the right amount of CPU cores for data preprocessing [5], [70]. While using our data preprocessing accelerator as a drop-in replacement for CPUs within the disaggregated CPU servers is a practically feasible option, we observe that such a design point is sub-optimal as it unnecessarily incurs high network traffic to copy data in (the raw data to be preprocessed) and out (the preprocessed train-ready tensors) of the disaggregated server for data preprocessing.

To this end, we present *PreSto* (**Pre**processing in-**Sto**rage), which is an In-Storage Processing (ISP) based data preprocessing system for RecSys training. In conventional systems, the petabyte-scale raw data that is to be preprocessed are stored in a distributed storage system. Rather than copying the raw data over the datacenter network and preprocessing them at a disaggregated, *remote* preprocessing server, *PreSto* conducts preprocessing *near data* using ISP. We demonstrate that such "storage-centric" data preprocessing can effectively close the performance gap between preprocessing and model training at a much lower cost compared to existing solutions. Overall, *PreSto* provides high speedup on data preprocessing performance (average 9.6$\times$) at low cost, significantly reducing the TCO (average 4.3$\times$) and energy consumption (average 11.3$\times$) vs. the baseline CPU-centric disaggregated preprocessing.

II. Background

A. End-to-End RecSys Training Pipeline

DNNs typically require some form of input data preprocessing before training. For instance, image classification requires preprocessing operations like image decoding, resizing, cropping, or augmentation. Additionally, speech recognition preprocessing includes Fourier transform and normalization of audio data [57]. Similarly, RecSys also requires a unique data preprocessing to generate the train-ready tensors consumed by the model training stage, which we detail below.

Traditionally, the RecSys training pipeline employed *offline* data preprocessing where the raw feature data stored in the storage system is transformed into train-ready tensors well before model training takes place. However, the proliferation of petabyte-scale data and the frequent development of diverse RecSys models by ML engineers have made offline preprocessing impractical because it is difficult to manage the substantial storage space required to store the offline preprocessed data. This challenge has triggered a shift towards *online* preprocessing [70], which involves preprocessing the raw feature data "on-the-fly" (Figure 1). The train-ready tensors are derived from the features that are constantly generated online by inference services. Specifically, inference servers log various end-user's interactions with the inference service as distinct features (e.g., news feed a user has clicked, items a user has purchased) using logging engines, e.g., Meta's Scribe [28]. Additionally, various streaming and batch engines, such as Spark [69], further label and filter data before storing them in a centralized data warehouse [72]. These raw feature data are categorized into two types: dense and sparse. Dense features represent continuous values (e.g., the time when a user viewed a video from YouTube), while sparse features represent sparse categorical values which can be variable-length (e.g., list of YouTube videos a user has viewed over a one-hour period). The logged features archived within the centralized data warehouse are then fetched into the storage system of each datacenter for future data preprocessing. The data preprocessing stage, also known as the ETL (**E**xtract, **T**ransform, and **L**oad) phase, involves the following series of operations:

- (Extract) The logged raw feature data are first retrieved from the storage system in preparation for the feature-specific transform operations.
- (Transform) The extracted raw feature data go through feature generation and feature normalization in order to generate the train-ready tensors.
- (Load) The train-ready tensors are copied over to the GPU's high-bandwidth memory (HBM) in preparation for the RecSys model training.

Once the train-ready tensors are loaded into GPU's memory, the actual model training is undertaken. Specifically, the GPU executes the embedding layers (embedding lookups, pooling for embedding reductions), feature interactions (batched GEMM), and MLP layers (GEMM). When it comes to embedding layers, the embedding look-up operation retrieves embeddings from an embedding table [51], which utilizes embedding indices that are generated during the Transform phase of ETL (e.g., the embedding indices transformed from feature X' and W in Figure 1 are utilized for embedding look-up operations). Recent work on various system-level performance optimizations for RecSys has primarily focused on this "training" stage of end-to-end training pipeline, rather than data preprocessing. The focus of this work is on RecSys data preprocessing, so we refer to these relevant prior studies for more details on the RecSys model architectures and training/inference [1], [4], [16]–[20], [22], [29], [30], [34]–[37], [48], [51], [56], [61], [66].

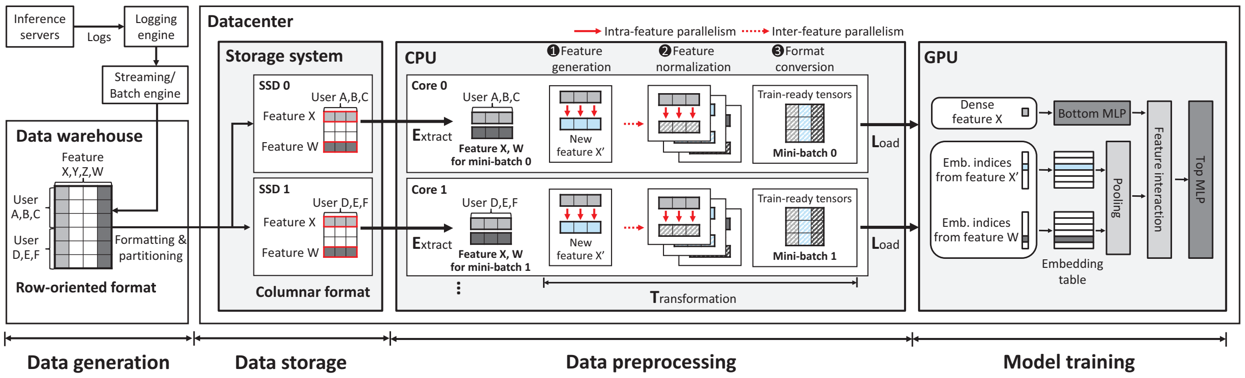

Fig. 1: High-level overview of the end-to-end RecSys training pipeline. In this work, we assume our baseline data storage and ingestion pipeline for data preprocessing by referring to the related academic literature published by Meta [28], [70]–[72].

While GPU-centric systems are popular options for training purposes, it is worth emphasizing that the entire data preprocessing stage (the ETL phase) is executed using CPUs in state-of-the-art data preprocessing systems [5], [14], [32], [47], [49], [65], including those for RecSys [25], [50], [70], [71]. In the rest of this paper, we assume such "CPU-centric" RecSys data preprocessing system for our baseline system.

B. Key Properties of RecSys Data Preprocessing

While numerous prior work explored preprocessing for vision, speech, and language processing [5], [8], [9], [14], [32], [33], [47], [49], [53], [65], data preprocessing for RecSys is relatively less explored [25], [50], [70], [71]. Unlike image or audio data, RecSys data is represented in a *tabular* format with multiple rows and columns. Concretely, each row represents an individual "user" whereas each column represents a distinct "feature" related to that user's past interactions with the RecSys inference service (shown as the data generation stage in Figure 1).

In the context of online preprocessing, deciding which features to utilize for model training depends on the ML engineer's choice, i.e., it is extremely challenging to predict which specific features will be utilized. Consequently, the hardware/software system for online preprocessing exhibits some unique properties in the Extract and Transform phase:

- (Extract) The raw feature data is first converted and stored in a *columnar* format (shown as the data storage stage in Figure 1). A group of rows are sharded into mutually exclusive *partitions* and different partitions are stored as independent columnar files into a distributed storage system of datacenter (e.g., the two partitions in Figure 1 are stored as two separate columnar files over two SSDs). The reason why these tabular data are converted in a columnar format is to avoid overfetching unwanted features. For example, in the data storage stage depicted in Figure 1, with a columnar format, any given feature for all the users can be *selectively* extracted from the storage system without having to retrieve unwanted features, e.g., it is possible to only fetch features X and W without having to fetch features Y and Z. In contrast, with the original, row-oriented format, extracting features X and W for all users inevitably leads to (unwanted) features Y and Z to be retrieved, wasting data read bandwidth.
- (Transform) The transformations conducted at this phase generate the mini-batch inputs that are utilized by the GPU for model training. All transformation operations performed within a mini-batch (detailed in the next subsection) are executed independently from transformations targeting other mini-batches. This is because RecSys transformation operations exhibit abundant *inter-/intra-feature parallelism*. Specifically, each element within a given feature vector (e.g., user A and C's feature X in Figure 1) represents a given user's interaction and there exists no data dependency across different users. Therefore, a transformation operation can be conducted element-wise by exploiting *intra*-feature parallelism. Similarly, different features (e.g., feature X and W for all users) are subject to independent transformation operations by leveraging *inter*-feature parallelism.

C. Feature Generation/Normalization in Data Preprocessing

RecSys data preprocessing can be divided into three key steps. First, the *feature generation* step generates new features using the raw feature data extracted from storage (Step ❶ in Figure 1, e.g., a new feature X' is generated from the raw feature X). Notably, one of the representative sparse feature generation operations is the "Bucketize" operation [63], [70], which transforms dense features into sparse features by sharding features based on predefined bucket boundaries (Algorithm 1[1]). Once the desired number of features is generated, they undergo *feature normalization* (Step ❷). Common feature normalization techniques include "Log" (which normalizes dense features using a logarithmic function) and "SigridHash" [63], [70] (which normalizes sparse features by computing a hash value that maps those features within the maximum index of the corresponding embedding table of the RecSys model, see Algorithm 2). Finally, the normalized features are converted

[1]This paper focuses on the feature generation (Bucketize) and feature normalization (SigridHash, Log) operations publicly available in the open-source TorchArrow [63]. Algorithm 1 and Algorithm 2 are simplified versions of each algorithm implemented in TorchArrow.

Algorithm 1 Bucketize for feature generation [63]

```
1: Input dense feature $a[1 \ldots n]$; bucket boundary $b[1 \ldots m]$;
   output $c[1 \ldots n]$
2: /* Digitize input dense features based on bucket */
3: for $i \leftarrow 1$ to $n$ do
4:     /* Find the index of buckets to which the input value
       belongs using binary search algorithm*/
5:     $c[i] \leftarrow \texttt{SearchBucketID}(a[i], b[1 \ldots m])$
6: end for
```

Algorithm 2 SigridHash for feature normalization [63]

```
1: Input sparse feature $a[1 \ldots n]$; seed $s$; max value $d$; output
   $c[1 \ldots n]$;
2: /* Apply hashing to input sparse features and limit their
   values */
3: for $i \leftarrow 1$ to $n$ do
4:     /* Compute seeded hash function */
5:     $h \leftarrow \texttt{ComputeHash}(a[i], s)$
6:     $c[i] \leftarrow h \bmod d$
7: end for
```

into an input mini-batch (Step ❸), which eventually gets loaded into GPU memory for training the RecSys model.

D. System Architecture for CPU-centric Data Preprocessing

Software architecture. As shown in Figure 1, the RecSys training pipeline employs the producer-consumer model. GPU training workers load and consume train-ready tensors (i.e., mini-batches) that the CPU preprocessing workers generate by transforming raw feature data. Because all transformation operations conducted within a mini-batch are executed locally without any dependencies to other mini-batches, state-of-the-art frameworks for end-to-end RecSys training pipeline such as TorchRec [24] allocate a single worker per each CPU core to handle the generation of each train-ready tensors that constitute a given mini-batch. By spawning multiple CPU workers in parallel, multiple input mini-batches are concurrently generated, enabling scalable improvements in data preprocessing throughput.

Hardware architecture. While the software architecture of RecSys data preprocessing provides high scalability, a critical challenge remains regarding *how to allocate a sufficient number of CPU cores that provide high enough data preprocessing throughput that meets the throughput demands of GPU-side training?* Traditionally, model training and data preprocessing jobs are co-located within the same server node containing multiple GPUs (Figure 2(a)) [47], [49]. As such, the performance of such co-located training pipeline is limited by how many CPU cores are available within the same server node (e.g., NVIDIA DGX system contains 8 A100 GPUs and 128 CPU cores, allowing a single GPU to utilize 16 CPU cores for data preprocessing), potentially suffering from significant GPU underutilization when the aggregate CPU-side data preprocessing throughput underwhelms the GPU-side training performance (detailed in Section III-A).

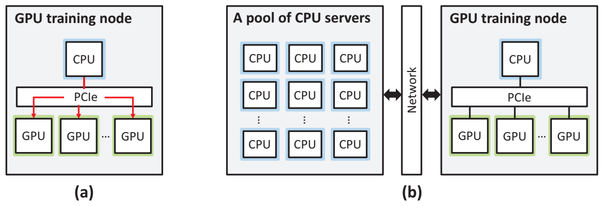

Fig. 2: System architectures for RecSys training. (a) A system that co-locates CPU-based data preprocessing workers with GPU-based model training workers within the same server node. (b) A system that provisions a pool of disaggregated CPU servers for data preprocessing.

To address these challenges, Zhao et al. [70] and Audibert et al. [5] proposed server disaggregation where a pool of CPU servers is reserved for data preprocessing (Figure 2(b)). Disaggregating CPU servers allows a large pool of CPU cores to be elastically allocated on-demand for preprocessing, effectively closing the performance gap between preprocessing and model training [25], [70]. However, such design point incurs substantial deployment cost and power consumption due to the large number of pooled CPU servers.

III. Characterization and Motivation

In this paper, we utilize the open-source RecSys data preprocessing library TorchArrow [63] to conduct a workload characterization study on state-of-the-art, CPU-centric data preprocessing systems. We note that there exists a significant disparity between the publicly available RecSys dataset [10] and the characteristics of a production-level dataset mentioned by a recent work from Meta [70]. Specifically, compared to the public dataset, production-level RecSys datasets contain a much larger number of dense/sparse features and larger average sparse feature length. Such discrepancy can undermine the primary objective of our study which is to characterize state-of-the-art RecSys preprocessing. Consequently, we scale up the open-sourced Criteo dataset [10] (referred to as RM1 in this paper) and develop four *synthetic* RecSys models (RM2-5) based on [70] to better represent the properties of production-level RecSys datasets. Table I summarizes the details of our public/synthetic datasets and the RecSys models trained. Our characterization is conducted with a training batch size of 8,192. Section V further details our evaluation methodology.

A. Motivation

As discussed in Section II-D, the aggregate data preprocessing throughput is strictly determined by how many CPU cores (i.e., the number of data preprocessing workers) are utilized for preprocessing. Consider a co-location based RecSys training system (Figure 2(a)) using a state-of-the-art DGX server [52] which contains 8 A100 GPUs and 128 CPU cores, allowing a single GPU to utilize 16 CPU cores for data preprocessing. In Figure 3, we scale up the number of CPU cores for preprocessing (from 1 to a maximum of 16) and study its effect on data preprocessing throughput (left axis) and the percentage of execution time the A100 GPU is actually training the model (right axis). We make the following two key observations from this experiment. First, the preprocessing

Type		Data preprocessing configuration parameters					RecSys model architecture			
		# Dense feats.	# Sparse feats.	Avg. sparse feat. length	# Generated sparse feats.	Bucket size	Bottom MLP	Top MLP	# Tables	Avg. # Embeddings
Public	RM1	13	26	1 (fixed)	13	1024	512-256-128	1024-1024-512-256-1	39	500,000
Synthetic	RM2	504	42	20	21	1024	512-256-128	1024-1024-512-256-1	63	500,000
	RM3	504	42	20	42	1024	512-256-128	1024-1024-512-256-1	84	500,000
	RM4	504	42	20	42	2048	512-256-128	1024-1024-512-256-1	84	500,000
	RM5	504	42	20	42	4096	512-256-128	1024-1024-512-256-1	84	500,000

TABLE I: The RecSys training dataset configuration and the target model architecture. RM1 is based on the public Criteo dataset [10] and RM2-5 are synthetically generated models we created by referring to production-grade RecSys dataset's characteristics released by Meta [70].

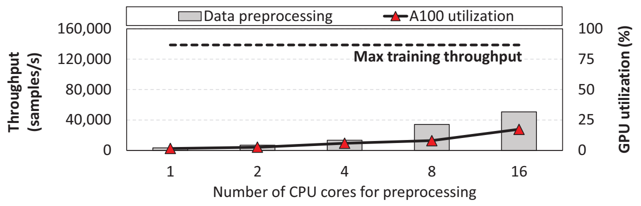

Fig. 3: Effective preprocessing throughput (left axis) and the resulting GPU utilization (right axis) as a function of the number of CPU cores (i.e., number of preprocessing workers) utilized for preprocessing. The dotted line shows the upperbound, maximum training throughput achievable using a single NVIDIA A100 GPU (left axis), which assumes the GPU is seamlessly fed with sufficient amount of train-ready tensors without interruption. To measure GPU's utilization, we use the CUDA Profiling Tools Interface (CUPTI) library. The experiment is collected over the evaluation platform detailed in Section V using our synthetic model RM5.

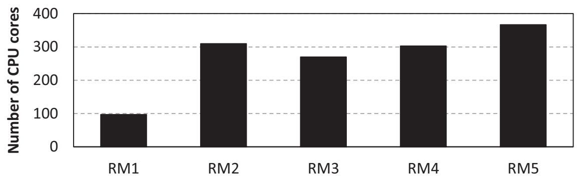

Fig. 4: The number of CPU cores required for CPU-centric preprocessing to fully utilize a training node containing 8 A100 GPUs.

throughput increases almost linearly as a function of the number of CPU cores, achieving 15× throughput improvement with 16 preprocessing workers vs. a single worker. Second, even with 16 preprocessing workers (i.e., the maximum number of CPU cores that can be allocated in co-located RecSys preprocessing), the GPU spends less than 20% of its execution time actually conducting model training as the train-ready tensors are not being sufficiently supplied to the GPU. Consequently, the end-to-end training throughput gets bounded by the effective preprocessing throughput which is well below the maximum training throughput achievable (dotted line).

Server disaggregation for data preprocessing [5], [25], [70] is an effective solution to close such wide performance gap, as it enables the allocation of any number of CPU preprocessing workers as required by the GPU training stage (Figure 2(b)). In Figure 4, we derive the number of CPU cores required in the data preprocessing stage to sustain the model training stage's high throughput requirement. For production-level synthetic datasets with large number of features and sparse feature lengths, several hundreds of CPU cores are required (367

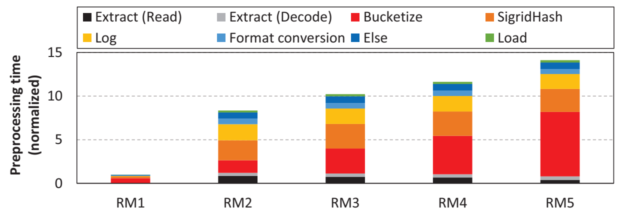

Fig. 5: Latency to preprocess a single mini-batch input using a single preprocessing worker in the baseline CPU-centric system, broken into key steps of preprocessing. The "Extract" stage (Section II-A) is further divided into (1) latency to fetch encoded raw feature data from the remote storage node (denoted as "Extract (Read)") and (2) latency spent decoding them (denoted as "Extract (Decode)"). As depicted, data preprocessing is bounded by the compute-intensive feature generation and normalization operations, rather than I/O operations ("Extract (Read)"). All results are normalized to RM1.

cores for RM5) to sufficiently supply the train-ready tensors for an 8 A100 GPU server node. It is important to note that hundreds to thousands of such production-level RecSys models are developed by ML engineers, invoking numerous concurrent training jobs executed over several tens of thousands of high-performance GPUs across the datacenter fleet [25], [70]. Such high demand for RecSys model training directly translates into substantial cost and power consumption in maintaining the disaggregated CPU servers for data preprocessing, e.g., Meta states that up to 60% of power consumption in RecSys training pipeline is dedicated to the storage and data ingestion pipeline of online CPU-centric data preprocessing [70].

Given such, a key motivation of this work is to conduct a detailed characterization on CPU-centric RecSys data preprocessing systems. In the remainder of this section, we root-cause the system-level bottlenecks of existing solutions in search for a scalable and cost-effective preprocessing system for RecSys.

B. Breakdown of End-to-End Data Preprocessing Time

To identify the key bottlenecks in RecSys data preprocessing, Figure 5 first evaluates end-to-end latency to preprocess a single training mini-batch during the ETL phase. Compared to the public RM1, the production-level RM2-5 contain a larger number of dense and sparse features with larger average sparse feature lengths. As such, these production-level models experience a substantial increase in total preprocessing time where RM5 experiences the largest 14× increase in latency. Specifically, with the larger number of features in the RM2-5 models, both dense and sparse feature normalization time

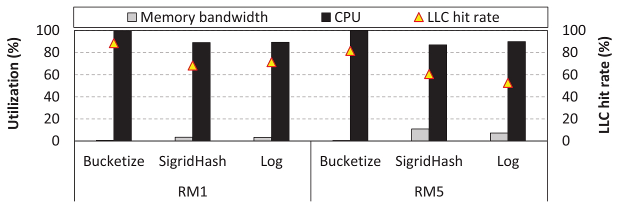

Fig. 6: CPU and memory bandwidth utilization (left) and LLC hit rate (right) during the execution of feature generation/normalization operations for the public model (RM1) and the production-scale model (RM5). We utilize Linux `perf` for our evaluation.

(i.e., Log and SigridHash, respectively) account for up to 55% of its preprocessing time. Additionally, these production-grade RM2-5 models experience a notable increase in feature generation time (i.e., Bucketize) due to the large number of sparse features to generate and its large bucket size. While RM3-5 all have the same number of sparse features to generate (at 42), larger bucket sizes (from 1024 to 4096 in RM3 to RM5, m in Algorithm 1) incur longer feature generation time because the latency to search the bucket ID of each feature value increases. Conversely, despite RM1-3 all having the same bucket size at 1024, the larger number of sparse features to generate (from 13 to 42 in RM1 to RM3) leads to larger feature generation time as well.

Overall, feature generation (Bucketize) and feature normalization (SigridHash and Log) collectively account for an average 79% of the data preprocessing time and become the most significant performance limiter in RecSys preprocessing.

C. Analysis on Performance-Limiting Operations

Figure 6 shows the CPU and memory bandwidth utilization and the last-level cache (LLC) hit rate during the execution of the three key performance-limiting operations (i.e., Bucketize, SigridHash, Log) in RM1 and RM5, respectively. Our analysis reveals the following key insights. First thing to note is that all three operations exhibit high CPU utilization with relatively low memory bandwidth utilization. RM5 in particular achieves increased memory bandwidth utilization as it has more features to generate and normalize compared to RM1. Nonetheless, the memory bandwidth utility of RM5 still remains under 15% of the maximum 281.6 GB/sec of memory throughput, exhibiting a compute-bound behavior. While the feature generation and normalization operations require large amount of data accesses, the actual working set it touches upon is relatively small. In RM1 and RM5, it amounts to as small as several tens of KBs to at most tens of MBs. For instance, the Bucketize operation involves sharding features based on the predefined bucket range whose active working set fits well within on-chip caches, leading to an LLC hit rate of 85%.

D. Our Goal: Scalable & Cost-Effective Preprocessing

Overall, we conclude that current CPU-centric RecSys preprocessing systems are bounded by the level of computation power available with CPUs, failing to fully reap out the inter-/intra-feature parallelism inherent in preprocessing. Although disaggregating a pool of CPUs for data preprocessing can help meet the computation demands of preprocessing, it requires high deployment cost and high power consumption. We argue that the abundant inter-/intra-feature parallelism in data preprocessing is well-suited for domain-specific acceleration, proposing an *accelerated* computing system for RecSys data preprocessing which is scalable and cost-effective.

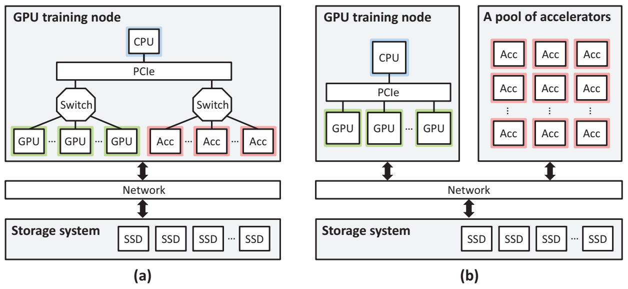

Fig. 7: (a) Preprocessing accelerators co-located with GPUs and (b) disaggregated pool of preprocessing accelerators.

IV. *PreSto*: An In-Storage "Pre"processing Architecture for RecSys Training

We present *PreSto* (**Pre**processing in-**Sto**rage), our In-Storage Processing (ISP) based data preprocessing system for RecSys training. Our proposed system offloads the time-consuming feature generation and normalization operations to an accelerator that is tightly coupled with the storage system.

A. System Design Considerations

Our key proposition is to employ accelerated computing for data preprocessing, so an important question to be answered is *where* our accelerator(s) should be deployed within the system architecture. Below we elaborate on the two possible design points that retrofit our data preprocessing accelerator within conventional co-located and disaggregated server architectures.

Preprocessing accelerators co-located with GPUs. Figure 7(a) illustrates our first design point, a scale-"up" server architecture employing a PCIe switch to co-locate the preprocessing accelerators with the GPUs. When the number of co-located accelerators is large enough to meet the GPU's training demands, this design point can obviate the need for disaggregated CPU servers dedicated to preprocessing. However, a critical limitation of such scale-up server is its *scalability* because the aggregate preprocessing throughput is still bounded by the total number of accelerators co-located within this scale-up server. As such, for large-scale RecSys models whose data preprocessing demands exceed the preprocessing throughput available with co-located accelerators, the GPU training workers will still suffer from idle periods. Another key concern with this approach is that both the accelerator-side data preprocessing workers and the GPU-side training workers all time-share the PCIe bus to communicate with the host CPU. Because preprocessing workers and training workers concurrently execute in a training pipeline, the PCIe bus can become a performance hotspot under such unbalanced system architecture. Given such critical limitations, we conclude that such scale-up solution is impractical.

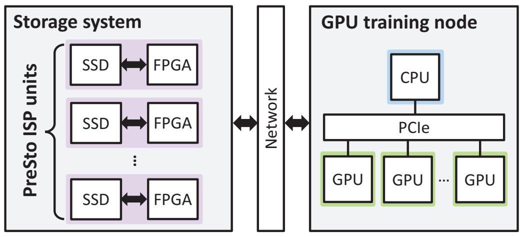

Fig. 8: System architecture of PreSto.

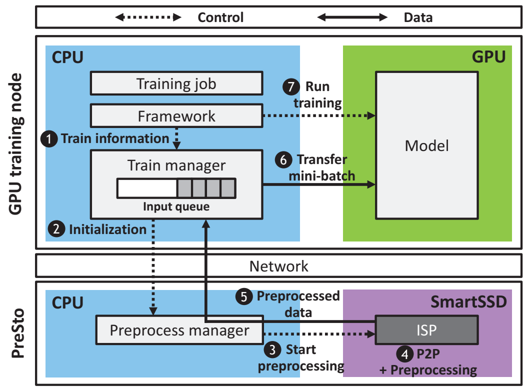

Fig. 9: Major components of *PreSto* software system and key steps undertaken during end-to-end RecSys training.

Disaggregated pool of preprocessing accelerators. To address the scalability issue in our scale-up server design, an alternative solution would be to utilize our preprocessing accelerator as a drop-in replacement of CPUs in the disaggregated preprocessing CPU pool. As shown in Figure 7(b), this design point effectively provides a disaggregated pool of preprocessing accelerators to the GPU training workers. By decoupling training jobs from preprocessing jobs over distinct server pools, the optimal number of preprocessing accelerators to allocate that meets a target training job's throughput demands can be determined dynamically, providing high scalability. Furthermore, this design point can better exploit inter-/intra-feature parallelism using domain-specific acceleration for high efficiency. However, similar to the baseline CPU-centric disaggregated preprocessing, server disaggregation still incurs substantial deployment cost and high TCO.

B. Proposed Approach: In-Storage Data Preprocessing

Hardware architecture. Due to the aforementioned challenges of co-located and disaggregated accelerator systems, our proposed system employs ISP (in-storage processing) architectures for data preprocessing, i.e., *in-storage "pre"processing*. As shown in Figure 8, this approach utilizes ISP devices like SmartSSDs [59] to directly replace conventional SSD cards. A SmartSSD tightly couples a normal SSD with a lightweight FPGA device within the NVMe U.2 form factor. This allows SmartSSDs to become a drop-in replacement for normal SSDs while still staying within the NVMe's 25 Watts power envelope[2]. As such, a SmartSSD can utilize its local FPGA device to implement our preprocessing accelerator right next to the local SSD where the raw feature data is stored. Such design point effectively tackles the system challenges of both co-located and disaggregated preprocessing accelerators as follows.

1) **Scalability.** As discussed in Figure 1, the tabular raw feature data subject for preprocessing is stored as columnar files. A group of rows within the tabular data is sharded into partitions and different partitions are stored as independent columnar files in a distributed storage system (e.g., 2 columnar files stored in two SSDs in Figure 1). While multiple blocks that constitute a single columnar file can further be distributed across multiple storage devices, state-of-the-art file systems for RecSys (e.g., Meta's Tectonic file system [55]) store all the blocks in a given partition contiguously within a single storage device. This ensures that all preprocessing operations can be conducted locally within a SmartSSD because all transformation operations conducted within a mini-batch (i.e., partition) are executed locally without any dependencies to other mini-batches (i.e., partitions in other columnar files). Consequently, in our proposed system, the overall preprocessing throughput can scale proportionally with the number of SmartSSDs allocated to a preprocessing worker targeting a given training job. A training job with a target preprocessing throughput in mind is first allocated with the appropriate number of SmartSSDs (detailed later in this subsection), one that is large enough to sustain the training stage's preprocessing need. We then have each SmartSSD independently extract raw feature data from its local SSD, which is immediately P2P transferred over to the local FPGA device for on-the-fly preprocessing. Because multiple SmartSSDs (i.e., multiple preprocessing workers) concurrently conduct preprocessing and generate mini-batch inputs, our *PreSto* ISP units provide highly scalable preprocessing service.
2) **Efficiency.** In our proposed system, all preprocessing operations are conducted *locally* within the storage system. This is because the SmartSSD's FPGA accelerator can extract the raw feature data directly from its local SSD using P2P data transfers, obviating the need for a disaggregated accelerator server pool with high maintenance cost. Such design decision also helps eliminate the communication overhead associated with disaggregated accelerator server designs (Figure 7(b)), which requires the raw feature data to be transferred from the storage system to the remote accelerator pool. It is also worth pointing out that SmartSSDs can seamlessly be deployed within the power constraints of the baseline distributed storage system, all thanks to the use of commodity devices that operate within the NVMe SSD's power budget (less than 25 watts per device [6]). Consequently, *PreSto* is minimally intrusive to existing hardware infrastructure while maintaining power-efficiency via accelerated computing.

Software architecture. Figure 9 provides a high-level overview of our software architecture. The two key components

[2]A high-end FPGA card like Xilinx U280 [67] which has a TDP of 225 Watts cannot be utilized for a U.2 SmartSSD card.

of our software system are the train manager and the preprocess manager. The train manager is implemented as part of the training worker process whose main role is to manage the end-to-end model training job, from data preprocessing to model training. That is, it requests the mini-batch inputs (i.e., train-ready tensors) from the storage system and once the mini-batch inputs are returned, they are forwarded to the GPU for model training. The preprocess manager is in charge of spawning and managing the actual preprocessing workers using the SmartSSD devices. Once the mini-batch inputs are ready, the preprocess manager returns them back to the train manager. Below we summarize the major steps undertaken during the end-to-end RecSys training process.

1) When a training job is launched by TorchRec [24], the train manager receives important information about the target training job (e.g., model configuration for training/preprocessing, mini-batch size, and other meta-data) and goes through several boot-strapping procedures in preparation for model training. These include the allocation of an input queue to store the mini-batch inputs designated for model training and a Remote Procedure Call (RPC) initialization (step ❶ in Figure 9).
2) Using the training job information, the train manager measures the maximum training throughput achievable with GPUs for the given training job. This is done by supplying the GPUs with dummy mini-batch inputs and stress-testing their highest sustainable throughput. This process only takes tens of seconds, the overhead of which is amortized over the several hours/days worth of training time. The train manager then initializes the preprocess manager, sending information relevant to the preprocessing jobs (e.g., configuration parameters of preprocessing). One of the important information that is forwarded to the preprocess manager is GPU's maximum training throughput (T), as it determines the level of preprocessing throughput that must be sustained in order to fully utilize the GPUs for model training. As such, the preprocess manager measures offline the maximum preprocessing throughput delivered with a single SmartSSD device under the given preprocessing configuration (P). By dividing the maximum training throughput with the per-SmartSSD preprocessing throughput (T/P), the preprocess manager derives the number of SmartSSD devices that need to be allocated to fully saturate the GPUs (step ❷).
3) The preprocess manager launches the necessary number of preprocessing workers based on the derived number of SmartSSD (T/P) (step ❸). As each preprocessing worker independently generates mini-batch inputs locally within the SmartSSD, each device fetches its share of raw feature data from the local SSD and transfers them P2P to the FPGA to conduct preprocessing on-the-fly (step ❹).
4) Once the mini-batch inputs are ready, they are converted into a train-ready tensor format, as required by TorchRec, and copied over to the input queue within the train manager (step ❺).

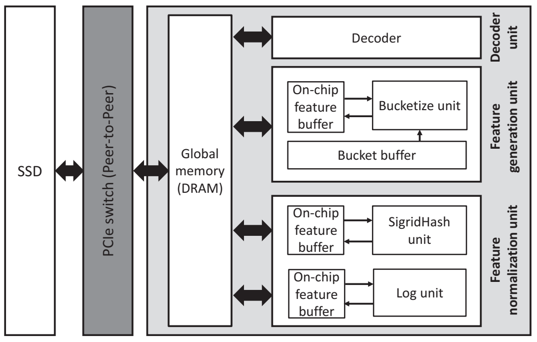

Fig. 10: *PreSto* accelerator microarchitecture.

5) Finally, the train manager transfers the mini-batch from its input queue to the GPU and kicks off the model training process (step ❻ and ❼). Because multiple SmartSSDs are concurrently preprocessing raw features and replenishing the train manager's input queue, it ensures that the GPU is constantly supplied with sufficient amount of mini-batch inputs to consume.

C. Data Preprocessing Accelerator Microarchitecture

The design objective of our *PreSto* accelerator is to maximally exploit inter/intra-feature parallelism inherent in RecSys data preprocessing. As depicted in Figure 10, we use the custom logic within the FPGA to implement a hardwired decoder unit, feature generation units, and feature normalization units. Each unit is equipped with the essential hardware logic tailored to the following operations: "Decoder" for columnar file decoding (our columnar files assume the Apache Parquet file format [3]), "Bucketize" for feature generation, and "SigridHash" as well as "Log" for feature normalization. To maximally exploit inter-/intra-feature parallelism, we employ the following design optimizations. First, to exploit inter-feature parallelism, we deploy multiple processing elements dedicated to each individual feature, directly connected to the off-chip-memory interface to fully utilize the bandwidth of global memory (DRAM). Second, to exploit intra-feature parallelism, each processing element employs double-buffering to overlap the next feature value's data fetch operation with the current feature value's generation and normalization operations. That is, once a portion of an input feature is fetched on-chip, its transformation is immediately executed while the next feature value is concurrently being fetched from the off-chip memory.

V. Methodology

A. Benchmarks

The majority of current academic research on RecSys utilizes the Criteo dataset [10], which is the largest publicly available dataset for RecSys. The Criteo dataset consists of 13 dense and 26 sparse features, with a fixed feature length of 1 for each sparse feature. According to recent work from Meta [70], production-level RecSys models can amount to 504 dense and 42 sparse features with an average sparse feature length of 20, much larger than the public Criteo dataset. To narrow this gap in our evaluation, we additionally construct four synthetic

Unit	LUT	REG	BRAM	URAM	DSP
Decode	18.84%	8.49%	25.08%	0.00%	0.00%
Bucketize	7.88%	4.28%	6.19%	27.59%	0.00%
SigridHash	23.11%	12.47%	11.89%	0.00%	19.19%
Log	4.18%	2.79%	4.89%	0.00%	10.62%
Total	54.02%	28.03%	48.05%	27.59%	29.81%

TABLE II: FPGA resource utilization of *PreSto*'s preprocessing accelerator. Decode, Bucketize, SigridHash, and Log units are synthesized with an operating frequency of 223MHz.

RecSys models in accordance with [70], expanding the existing features of the Criteo dataset to better cover the evaluation space of production-level RecSys models with large number of dense/sparse features. Table I details the configuration of the five RecSys models and their training datasets we evaluate.

B. Experimental Setup

Hardware. Exploring *PreSto* in a production-level training pipeline that accurately reflects industry's large-scale disaggregated CPU servers, multi-node/multi-GPU systems, and a distributed storage array integrated with SmartSSDs is challenging at the academic research level for several reasons. Aside from many undisclosed details of hyperscaler's production ML infrastructure, the unavailability of SmartSSDs in cloud services like Amazon AWS [2] rendered our experiments to employ the following methodology. We first demonstrate *PreSto*'s advantages in real systems by constructing a small-scale, proof-of-concept (PoC) prototype of *PreSto* using commodity CPU/GPU/SmartSSD devices. We then develop an analytical model that estimates *PreSto*'s performance at large-scale by utilizing real measurements from our PoC prototype.

At a high-level, our PoC prototype includes three major components: (1) a storage node (with and without a SmartSSD to model *PreSto* (with SmartSSD) and baseline storage system (without SmartSSD)), (2) a GPU training node, and (3) a pool of multiple CPU nodes for preprocessing (to model baseline disaggregated CPU preprocessing service), which communicate over a network using 10 Gbps Ethernet. Both the storage node and the pool of CPU nodes for preprocessing are designed using a total of three two-socket Intel Xeon Gold 6242 CPU nodes (32 CPU cores per node) where one node is used as the storage node and the other two nodes are utilized as a remote pool for data preprocessing (maximum $2\times32=64$ CPU cores available for data preprocessing). The GPU training node contains AMD EPYC 7502 CPU connected with a single NVIDIA A100 GPU. When evaluating *PreSto*, we add a single SmartSSD card [59] to the storage node which handles data preprocessing locally within the storage node. *PreSto*'s preprocessing accelerator is designed using Xilinx Vitis HLS 2022.2 whose resource utilization is summarized in Table II.

As for the analytical model we developed for large-scale performance estimations, we utilize the observations made from our characterization study in Section III where data preprocessing operations are embarrassingly parallel and exhibit high scalability. Because end-to-end training performance as well as its data preprocessing performance is throughput-bound rather than latency-bound, our analytical performance model assumes that the preprocessing throughput measured from our real PoC prototype (i.e., the preprocessing throughput measured with a single CPU core (baseline) and a single SmartSSD (*PreSto*)) scales proportionally with the number of CPU cores allocated (baseline) or the number of SmartSSDs allocated (*PreSto*) for data preprocessing.

Software. Our end-to-end RecSys training pipeline is implemented using TorchArrow (v0.1.0) [63] for data preprocessing and TorchRec (v0.3.2) [24] for model training, assuming a mini-batch size of 8,192. We assume the Apache Parquet file format [3] when the columnar raw feature data is stored in our storage system. Both baseline CPU-centric and *PreSto* preprocessing system communicate with the GPU training node using the PyTorch RPC API [11]. We use Xilinx Runtime library to manage *PreSto*'s FPGA device.

C. Evaluation Methods

Power measurement. When measuring the power consumption of the CPU-based storage node as well as the disaggregated preprocessing nodes, we measure its system-level power consumption using Intel Performance Counter Monitor (PCM) [23]. The Xilinx Vivado [68] and NVIDIA System Management Interface (nvidia-smi) are used when measuring the power of *PreSto*'s FPGA accelerator and the GPU, respectively.

Cost-efficiency. To quantify the cost-effectiveness of *PreSto*, we also evaluate cost-efficiency using the evaluation metric suggested in [42], [43] as summarized below:

$$\text{Cost-efficiency} = \frac{Throughput \times Duration}{\text{CapEx} + \text{OpEx}}$$

$$\text{where OpEx} = \sum(Power \times Duration \times Electricity)$$

CapEx (\$) refers to the one-time capital expenditure required to purchase and establish the hardware platform components. OpEx (\$) represents the operating expenditure of this hardware platform. To determine CapEx, we utilize the cost information obtained from the respective company website [12], [59]. OpEx is derived using the power consumed by the hardware components (*Power*), the active duration of each hardware component (*Duration*, a period of 3 years [7], [43]), and the average price of electricity (*Electricity*, \$0.0733/kWh [42], [43]). It is worth pointing out that the numerator value to calculate cost-efficiency ($Throughput \times Duration$) is identical for both baseline disaggregated CPU preprocessing and *PreSto*: both baseline and our proposal can sustain the throughput demands of GPU's training stage, so $Throughput$ and $Duration$ are constant values. Therefore, the difference in cost-efficiency is determined by (CapEx+OpEx).

VI. Evaluation

We first demonstrate *PreSto*'s merits using our PoC prototype (Section VI-A). We then utilize our analytical model (Section V) to estimate *PreSto*'s effect on performance, energy-efficiency, and TCO at larger scale (Section VI-B). In the

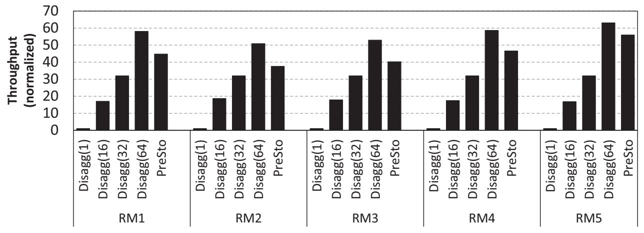

Fig. 11: Preprocessing throughput of *PreSto* (single SmartSSD) vs. *Disagg*. Disagg(*N*) is a design point executing with *N* preprocessing workers using *N* CPU cores. Results are normalized to Disagg(1).

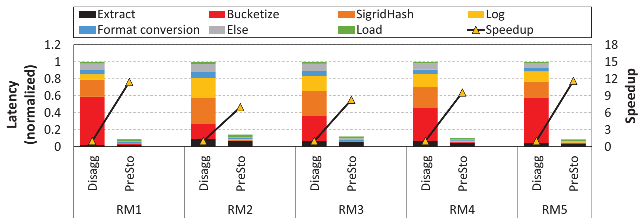

Fig. 12: (Left axis) Data preprocessing time to generate a single mini-batch using a single preprocessing worker. Latency is broken down into key steps of data preprocessing. (Right axis) *PreSto*'s end-to-end speedup for preprocessing. All results are normalized to *Disagg*.

rest of this section, the baseline CPU-centric preprocessing system assumes the disaggregated CPU server design (denoted "*Disagg*"), one which is equipped with a maximum of 64 CPU cores in our small-scale PoC prototype.

A. Performance and Cost-Effectiveness of PreSto (PoC)

Throughput. Figure 11 compares the data preprocessing throughput of *PreSto* vs. *Disagg*. As depicted, a single SmartSSD device (*PreSto*) consistently outperforms *Disagg* even with 32 CPU cores (i.e., a single CPU node) and demonstrates the benefits of our ISP solution. Since *Disagg*'s performance scales well to the number of CPU cores (workers) utilized, allocating more CPU cores can still match the throughput provided with *PreSto* albeit at a proportional increase in cost (e.g, Disagg(64) using two CPU nodes (64 cores) is able to slightly outperform *PreSto* by average 27% but with 2× higher cost).

Latency. To better highlight where *PreSto*'s speedup comes from, Figure 12 compares the latency to generate a single mini-batch input using a single preprocessing worker using *Disagg* and *PreSto*. The latency breakdown focuses on the key steps undertaken during preprocessing. In the baseline *Disagg*, the "Extract" step includes time to fetch encoded raw feature data from the remote storage node and decode them. With *PreSto*, the "Extract" step includes the P2P data transfer of the encoded raw feature data from local SSD to the FPGA which is immediately followed by their decoding using our dedicated decoder unit. Because the decoding algorithm is less parallelizable than feature generation and normalization operations, the reduction in the "Extract" step's execution time is less pronounced, rendering this step to account for an average 40.8% of the total preprocessing time of *PreSto*. Nonetheless, *PreSto* provides significant improvements in the performance of

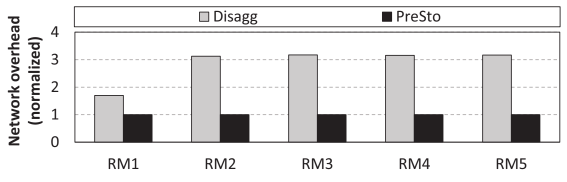

Fig. 13: Aggregate latency incurred during any RPC calls executed for inter-node communication during the course of data preprocessing.

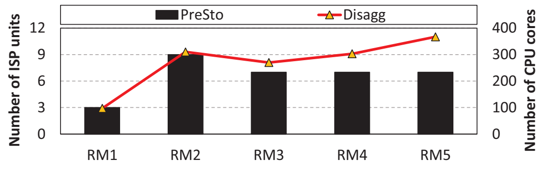

Fig. 14: The number of ISP units (left axis) and CPU cores (right axis) required for *PreSto* and *Disagg* to sustain a single multi-GPU server node containing 8 A100 GPUs.

feature generation (Bucketize) and normalization (SigridHash, Log), achieving an average 9.6× (maximum 11.6×) reduction in end-to-end preprocessing time. These results highlight the benefits of *PreSto*'s domain-specific acceleration using RecSys preprocessing's inter-/intra-feature parallelism.

Data movements. Another key benefit provided with *PreSto* is that all data preprocessing operations are conducted locally within the storage node, unlike *Disagg* which needs to explicitly copy data in (the raw data to be preprocessed) and out (the train-ready tensors) of the disaggregated CPU nodes for data preprocessing. Because our small-scale PoC prototype evaluates a single training job in a highly controlled, isolated setting, *Disagg*'s RPC communication time to read out the raw feature data from the remote storage node and copy into the disaggregated CPU nodes accounts for a relatively small portion of the end-to-end preprocessing time (but still accounting for 9.1% in RM2 under *Disagg* in Figure 12). Since real-world datacenter fleets concurrently handle a large number of training jobs, all of which time-share the datacenter network, *PreSto*'s ISP capability can be beneficial in alleviating the preprocessing operation's pressure on network communications. In Figure 13, we show the aggregate latency incurred during any RPC calls executed for inter-node data movements during the course of data preprocessing. Unlike *Disagg* which incurs additional latency when the preprocessing worker copies raw feature data from the remote storage to the disaggregated CPU nodes, our *PreSto* can completely eliminate such performance overhead. This leads *PreSto* to provide a 2.9× reduction in RPC-invoked inter-node communication time.

B. PreSto's Effect on Energy-Efficiency and TCO

We now evaluate *PreSto*'s effect on performance/Watt (energy-efficiency) and performance/$ (TCO) by utilizing our analytical model for large-scale experiments (Section V-B).

Energy-efficiency. As discussed in Figure 4, baseline *Disagg* requires significant number of CPU cores for data preprocessing (e.g., 367 cores for RM5), which translates into significant power consumption and high deployment cost. To

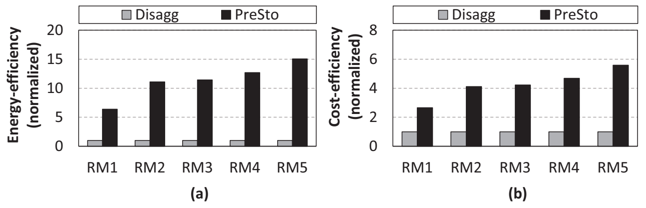

Fig. 15: (a) Energy-efficiency and (b) cost-efficiency. Power consumption is measured using Intel's Performance Counter Monitor (PCM) and Xilinx Vivado. Section V-C details our methodology.

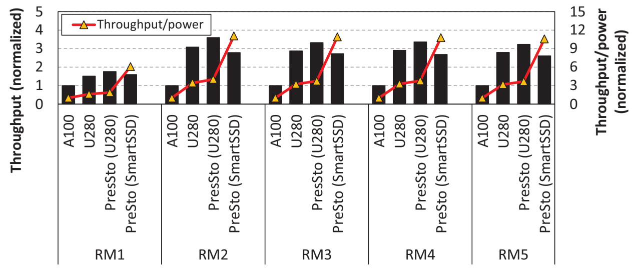

Fig. 16: Data preprocessing's (left axis) performance and (right axis) performance/Watt over *PreSto* (single SmartSSD device), *PreSto* (single U280 FPGA), a single A100 GPU and a single U280 FPGA.

quantitatively demonstrate *PreSto*'s effectiveness in energy and cost reduction, we evaluate how many ISP units (i.e., the number of SmartSSD cards) are required to match the preprocessing demands of a multi-GPU server containing 8 GPUs (Figure 14). Remarkably, to match such high GPU training throughput demand, *PreSto* only requires a maximum of 9 ISP units which incur (9×25)=225 Watts of worst-case power consumption (25 Watts TDP per each SmartSSD card). *Disagg*, on the other hand, requires up to 367 CPU cores (i.e., 12 CPU server nodes) to match *PreSto*'s preprocessing performance, incurring much higher power consumption as well as cost. Figure 15(a) summarizes how all of this translate into energy consumption. Overall, *PreSto* provides an average 11.3× (maximum 15.1×) energy-efficiency improvement, demonstrating its merits.

Cost-efficiency (TCO). Figure 15(b) compares the cost-efficiency of *PreSto* and *Disagg* for data preprocessing (as defined in Section V-C). Overall, *PreSto* provides an average 4.3× (maximum 5.6×) improvement in cost-efficiency vs. *Disagg*. Cost-efficiency is primarily determined by both CapEx and OpEx and our experiments thus far have demonstrated that *PreSto* outperforms *Disagg* on both fronts, providing significant reduction in TCO.

C. PreSto vs. Alternative Accelerated Preprocessing

Our study thus far have focused on comparing data preprocessing over the baseline disaggregated CPU servers and ISP architectures. For the completeness of our study, we also evaluate the efficacy of alternative accelerated preprocessing solutions where high-end GPUs/FPGAs function as preprocessing accelerators. Figure 16 compares the preprocessing performance (left axis) and performance/Watt (energy-efficiency) with four system design points: (1) a single A100 GPU (denoted "A100") and (2) a single Xilinx U280 [67] FPGA (denoted "U280") employed within a disaggregated accelerator pool (Figure 7(b)), as well as our proposed FPGA-based accelerator system implemented using (3) a single, discrete U280 FPGA card integrated within the storage node over PCIe (denoted "*PreSto* (U280)") and (4) one implemented using a single SmartSSD device (denoted "*PreSto* (SmartSSD)"). We utilizes NVIDIA's NVTabular library [54] for GPU-based data preprocessing. The U280-based FPGA accelerator is synthesized with 2× larger number of Decoder, Feature generation, and Feature normalization units that maximally utilize U280's larger custom logics. *PreSto* (SmartSSD) provides an average 2.5× speedup vs. A100 while experiencing an average 5% performance loss vs. U280 FPGA. Note that *PreSto* (SmartSSD) is able to achieve such performance despite its much lower power consumption (TDP of 25 Watts (SmartSSD) vs. 250 Watts (A100) and 225 Watts (U280)). In general, GPUs are throughput-optimized devices with high compute and memory throughput, so they perform best when the target application requires massive compute and memory accesses. We observe that the compute and memory operations entailed in RecSys preprocessing is lightweight vs. training. This makes it challenging for the GPU to amortize the cost of CUDA kernel launches, each of which has a small working set with modest compute/memory operations, leading to significant GPU underutilization. U280 does much better than the GPU but its end-to-end speedup vs. *PreSto* (SmartSSD) is relatively low, due to its high latency overhead in copying data in/out of the disaggregated preprocessing node (which accounts for an average 47.6% of its end-to-end preprocessing time). Even though *PreSto* (U280) minimizes such redundant data movements and achieves a slightly higher preprocessing throughput compared to *PreSto* (SmartSSD), *PreSto* (SmartSSD) delivers much higher energy-efficiency (an average 2.9×) vs. *PreSto* (U280) by being custom-designed to right-size its compute units for data preprocessing under a tighter power budget (25 Watts).

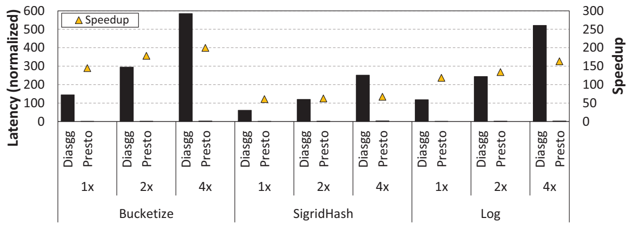

Fig. 17: Latency of *Disagg* and *PreSto*'s feature generation/normalization time (left axis) and *PreSto*'s speedup (right axis) when the the number of features for preprocessing is changed. The "1×" data point corresponds to the RM5 configuration in Table I. *Disagg*'s latency to conduct each operation is normalized to its respective "1×" latency of *PreSto*.

D. PreSto Sensitivity to the Number of Features to Preprocess

We investigate *PreSto*'s sensitivity to the number of features to preprocess by evaluating the latency incurred in executing the Bucketize, SigridHash, and Log operations when the number of generated, sparse, and dense features are changed. Figure 17 illustrates a comparison of the average latency to execute each operation using *Disagg* and *PreSto*. While the latency of *Disagg* increases almost proportionally with the number of features for preprocessing, *PreSto* does a much better job leveraging inter-/intra-feature parallelism and consistently achieves significant speedups, demonstrating the robustness of our proposal.

VII. Related work

There is a large body of prior literature exploring DNN preprocessing, RecSys model training/inference, RecSys data storage and preprocessing, and domain-specific/general-purpose ISP designs, which we summarize below.

DNN preprocessing. There are multiple prior work addressing the performance gap between DNN model training and data preprocessing [5], [8], [9], [13], [14], [32], [33], [47], [49], [53], [57], [65]. Mohan et al. [47] and Murray et al. [49] proposed software optimizations to address this problem, while TrainBox [57] and DALI [53] proposed hardware accelerated preprocessing tackling computer vision and audio training tasks. Similarly, DLBooster [9] focused on offloading preprocessing operations to an FPGA for inference. PreGNN [13] proposed offloading graph neural network (GNN) preprocessing tasks to an accelerator. Several prior art [5], [14], [32], [33], [65] proposed disaggregated preprocessing solutions but these works primarily focused on efficiently managing CPU resources via software optimizations using data caching or prefetching. Importantly, all of these prior art strictly do not focus on RecSys, rendering the key contribution of our work unique.

RecSys data storage and preprocessing. Zhao et al. [70] discusses a disaggregated data preprocessing service for Meta's RecSys training pipeline. XDL [25] proposed a distributed ML framework for Alibaba's production RecSys model. InTune [50] presents a reinforcement learning-based RecSys data pipeline optimization to efficiently manage CPU resources. There also exists prior work exploring feature deduplication to improve the performance of RecSys data preprocessing [71]. While not targeting the preprocessing stage of RecSys, Tectonic-shift [72] explored the viability of a flash storage tier in the data storage system of Meta's production ML training infrastructure, aiming to improve the performance and power efficiency of their I/O operation in data storage stage. Overall, the contribution of *PreSto* is orthogonal to these studies.

RecSys model training/inference. The surge of interest in RecSys in both academia and industry has spawned numerous prior work accelerating RecSys training and inference utilizing near-/in-memory processing [4], [29], [34], [35], [56] as well as various hardware/software optimizations [1], [16]–[20], [22], [30], [36], [37], [48], [51]. Importantly, *PreSto* stands apart from this body of work by focusing on accelerating RecSys preprocessing, which is as discussed in this paper completely orthogonal operation compared to model training/inference.

Domain-specific/general-purpose ISP designs. There is a large body of prior literature exploring domain-specific/general-purpose ISP designs. GLIST [39] proposed an ISP architecture for SSD-based GNN inference. SmartSAGE [38] proposed an ISP-based GNN training system to overcome I/O bottleneck of SSD-based training. Mahapatra et al. [43] proposed an ASIC-based ISP in a disaggregated storage system for serverless functions, alleviating communication overhead of remote storage systems. RecSSD [66] and RM-SSD [61] proposed ISP to overcome the memory bottlenecks of RecSys inference. GraphSSD [46] proposed an ISP architecture for graph semantics with a simple programming interface. ECSSD [40] proposed an ISP architecture for extreme classification based on the approximate screening algorithm. Work by Hu et al. [21] proposed an ISP-based dynamic multi-resolution storage system to mitigate the performance bottleneck of data preparation for approximate compute kernels. ASSASIN [73], INSPIRE [41], and GenStore [45] proposed an ISP architecture for stream computing, private information retrieval, and genome sequence analysis, respectively. There also exists a large body of prior literature targeting ISP acceleration for data-intensive workloads. Morpheus [64], DeepStore [44], Active Flash [62], GraFBoost [27], Biscuit [15], and BlueDBM [26] proposed a domain-specific ISP architecture that targets data analytics, data management, object (de)serialization, or graph analytics. Summarizer [31] and INSIDER [58] proposed a hardware/software co-designed ISP architecture to offload data-intensive tasks with a set of flexible programming APIs. Willow [60] proposed architectural support to enhance the effectiveness and flexibility of a programmable ISP design targeting I/O-intensive applications. Unlike these prior work, PreSto demonstrates the merits of an ISP solution targeting *compute-bound* RecSys data preprocessing and uncovers its new system-level bottlenecks, rendering our key contributions unique.

VIII. Conclusion

In this work, we propose an ISP based RecSys data preprocessing system called *PreSto* which conducts the preprocessing operation close to where the training samples are preserved. By fully leveraging inter-/intra-feature parallelism available in feature generation/normalization, *PreSto* can effectively close the performance gap between preprocessing and model training at a much lower cost and power consumption compared to the baseline CPU-centric system. Overall, *PreSto* outperforms state-of-the-art preprocessing systems with 9.6$\times$ speedup in end-to-end preprocessing time, 4.3$\times$ improvement in cost-efficiency, and 11.3$\times$ enhancement in energy-efficiency.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1A2C2091753), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2022-0-01037, Development of High Performance Processing-in-Memory Technology based on DRAM). We also appreciate the support from SNU-SK Hynix Solution Research Center (S3RC) and the EDA tool supported by the IC Design Education Center(IDEC), Korea. Minsoo Rhu is the corresponding author.

REFERENCES

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding Training Efficiency of Deep Learning Recommendation Models at Scale," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[2] Amazon, "Amazon AWS Cloud Computing Services," https://aws.amazon.com/, 2023.

[3] Apache Software Foundation, "Apache Parquet," https://parquet.apache.org/, 2023.

[4] B. Asgari, R. Hadidi, J. Cao, D. E. Shim, S.-K. Lim, and H. Kim, "FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[5] A. Audibert, Y. Chen, D. Graur, A. Klimovic, J. Šimša, and C. A. Thekkath, "tf.data service: A Case for Disaggregating ML Input Data Processing," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2023.

[6] A. Barbalace and J. Do, "Computational Storage: Where Are We Today?" in *11th Annual Conference on Innovative Data Systems Research (CIDR)*, 2021.

[7] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Springer Nature, 2019.

[8] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X.-H. Sun, and G. Chen, "iCache: An Importance-Sampling-Informed Cache for Accelerating I/O-Bound DNN Model Training," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[9] Y. Cheng, D. Li, Z. Guo, B. Jiang, J. Lin, X. Fan, J. Geng, X. Yu, W. Bai, L. Qu, R. Shu, P. Cheng, Y. Xiong, and J. Wu, "DLBooster: Boosting End-to-End Deep Learning Workflows with Offloading Data Preprocessing Pipelines," in *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019.

[10] Criteo, "Criteo Terabyte Click Logs," https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2013.

[11] P. Damania, S. Li, A. Desmaison, A. Azzolini, B. Vaughan, E. Yang, G. Chanan, G. J. Chen, H. Jia, H. Huang, J. Spisak, L. Wehrstedt, L. Hosseini, M. Krishnan, O. Salpekar, P. Belevich, R. Varma, S. Gera, W. Liang, S. Xu, S. Chintala, C. He, A. Ziashahabi, S. Avestimehr, A. Jain, and Z. DeVito, "PyTorch RPC: Distributed Deep Learning Built on Tensor-Optimized Remote Procedure Calls," in *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[12] Dell, "Dell R640," https://www.dell.com/en-us/dt/servers/poweredge-rack-servers.htm.

[13] D. Gouk, S. Kang, M. Kwon, J. Jang, H. Choi, S. Lee, and M. Jung, "PreGNN: Hardware Acceleration to Take Preprocessing Off the Critical Path in Graph Neural Networks," in *IEEE Computer Architecture Letters*, 2022.

[14] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, "Cachew: Machine Learning Input Data Processing as a Service," in *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2022.

[15] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[16] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A System for Optimizing End-to-End At-Scale Neural Recommendation Inference," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[17] U. Gupta, S. Hsia, J. Zhang, M. Wilkening, J. Pombra, H.-H. S. Lee, G.-Y. Wei, C.-J. Wu, and D. Brooks, "RecPipe: Co-Designing Models and Hardware to Jointly Optimize Recommendation Quality and Performance," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.

[18] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[19] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[20] S. Hsia, U. Gupta, B. Acun, N. Ardalani, P. Zhong, G.-Y. Wei, D. Brooks, and C.-J. Wu, "MP-Rec: Hardware-Software Co-design to Enable Multi-Path Recommendation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[21] Y.-C. Hu, M. T. Lokhandwala, T. I, and H.-W. Tseng, "Dynamic Multi-Resolution Data Storage," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[22] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A Chiplet-Based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[23] Intel, "Intel Performance Counter Monitor," https://github.com/intel/pcm, 2022.

[24] D. Ivchenko, D. Van Der Staay, C. Taylor, X. Liu, W. Feng, R. Kindi, A. Sudarshan, and S. Sefati, "TorchRec: a PyTorch Domain Library for Recommendation Systems," in *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2022.

[25] B. Jiang, C. Deng, H. Yi, Z. Hu, G. Zhou, Y. Zheng, S. Huang, X. Guo, D. Wang, Y. Song, L. Zhao, Z. Wang, P. Sun, Y. Zhang, D. Zhang, J. Li, J. Xu, X. Zhu, and K. Gai, "XDL: An Industrial Deep Learning Framework for High-dimensional Sparse Data," in *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019.

[26] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An Appliance for Big Data Analytics," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[27] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFBoost: Using Accelerated Flash Storage for External Graph Analytics," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[28] M. Karpathiotakis, D. Wernli, and M. Stojanovic, "Scribe: Transporting Petabytes per Hour via a Distributed, Buffered Queueing System," https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/, 2019.

[29] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[30] L. Ke, U. Gupta, M. Hempstead, C.-J. Wu, H.-H. S. Lee, and X. Zhang, "Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[31] G. Koo, K. K. Matam, T. I., H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading Communication with Computing Near Storage," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[32] M. Kuchnik, A. Klimovic, J. Simsa, V. Smith, and G. Amvrosiadis, "Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines," in *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[33] A. V. Kumar and M. Sivathanu, "Quiver: An Informed Storage Cache for Deep Learning," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[34] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[35] Y. Kwon, Y. Lee, and M. Rhu, "Tensor Casting: Co-Designing Algorithm-Architecture for Personalized Recommendation Training," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[36] Y. Kwon and M. Rhu, "Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

[37] Y. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "MERCI: Efficient Embedding Reduction on Commodity Hardware via Sub-Query Memoization," in *Proceedings of the International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021.

[38] Y. Lee, J. Chung, and M. Rhu, "SmartSAGE: Training Large-scale Graph Neural Networks using In-Storage Processing Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

[39] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "GLIST: Towards In-Storage Graph Learning," in *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2021.

[40] S. Li, F. Tu, L. Liu, J. Lin, Z. Wang, Y. Kang, Y. Ding, and Y. Xie, "ECSSD: Hardware/Data Layout Co-Designed In-Storage-Computing Architecture for Extreme Classification," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2023.

[41] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "INSPIRE: IN-Storage Private Information REtrieval via Protocol and Architecture Co-design," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

[42] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers," in *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2019.

[43] R. Mahapatra, S. Ghodrati, B. H. Ahn, S. Kinzer, S. ting Wang, H. Xu, L. Karthikeyan, H. Sharma, A. Yazdanbakhsh, M. Alian, and H. Esmaeilzadeh, "Domain-Specific Computational Storage for Serverless Computing," *arXiv preprint arXiv:2303.03483*, 2023.

[44] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "DeepStore: In-Storage Acceleration for Intelligent Queries," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[45] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "GenStore: A High-Performance In-Storage Processing System for Genome Sequence Analysis," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[46] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD: Graph Semantics Aware SSD," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

[47] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and Mitigating Data Stalls in DNN Training," in *Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[48] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

[49] D. G. Murray, J. Simsa, A. Klimovic, and I. Indyk, "tf.data: A Machine Learning Data Processing Framework," in *Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[50] K. Nagrecha, L. Liu, P. Delgado, and P. Padmanabhan, "InTune: Reinforcement Learning-Based Data Pipeline Optimization for Deep Recommendation Models," in *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2023.

[51] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv preprint arXiv:1906.00091*, 2019.

[52] NVIDIA, "NVIDIA DGX A100," https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[53] NVIDIA, "NVIDIA Data Loading Library," https://developer.nvidia.com/dali, 2023.

[54] NVTabular, https://github.com/NVIDIA-Merlin/NVTabular, 2023.

[55] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. Tipton, E. Katz-Bassett, and W. Lloyd, "Facebook's Tectonic Filesystem: Efficiency from Exascale," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[56] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. H. Ahn, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.

[57] P. Park, H. Jeong, and J. Kim, "TrainBox: An Extreme-Scale Neural Network Training Server Architecture by Systematically Balancing Operations," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020.

[58] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive," in *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2019.

[59] Samsung, "Samsung SmartSSD," https://samsungsemiconductor-us.com/smartssd/.

[60] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User-Programmable SSD," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[61] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[62] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[63] TorchArrow, https://github.com/pytorch/torcharrow, 2022.

[64] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[65] T. Um, B. Oh, B. Seo, M. Kweun, G. Kim, and W.-Y. Lee, "FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline," in *Proceedings of the VLDB Endowment (PVLDB)*, 2023.

[66] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "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)*, 2021.

[67] Xilinx, "Alveo U280 Data Center Accelerator Card," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[68] Xilinx, "Xilinx Vitis," https://www.xilinx.com/products/design-tools/vitis.html/, 2022.

[69] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[70] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C.-J. Wu, C. Kozyrakis, and P. Pol, "Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

[71] M. Zhao, D. Choudhary, D. Tyagi, A. Somani, M. Kaplan, S.-H. Lin, S. Pumma, J. Park, A. Basant, N. Agarwal, C.-J. Wu, and C. Kozyrakis, "RecD: Deduplication for End-to-End Deep Learning Recommendation Model Training Infrastructure," in *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[72] M. Zhao, S. Pan, N. Agarwal, Z. Wen, D. Xu, A. Natarajan, P. Kumar, S. S. P, R. Tijoriwala, K. Asher, H. Wu, A. Basant, D. Ford, D. David, N. Yigitbasi, P. Singh, and C.-J. Wu, "Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training," in *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2023.

[73] C. Zou and A. A. Chien, "ASSASIN: Architecture Support for Stream Computing to Accelerate Computational Storage," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022.