NDRec: A Near-Data Processing System for Training Large-Scale Recommendation Models

Shiyu Li, *Graduate Student Member, IEEE*, Yitu Wang, Edward Hanson, *Graduate Student Member, IEEE*, Andrew Chang, *Member, IEEE*, Yang Seok Ki, *Member, IEEE*, Hai Li, *Fellow, IEEE*, and Yiran Chen, *Fellow, IEEE*

***Abstract*—Recent advances in deep neural networks (DNNs) have enabled highly effective recommendation models for diverse web services. In such DNN-based recommendation models, the embedding layer comprises the majority of model parameters. As these models scale rapidly, the embedding layer's memory capacity and bandwidth requirements threaten to exceed the limits of current computing architectures. We observe the embedding layer's computational demands increase much more slowly than its storage needs, suggesting an opportunity to offload embeddings to storage hardware. In this work, we present *NDRec*, a near-data processing system to train large-scale recommendation models. NDRec offloads both the parameters and the computation of the embedding layer to computational storage devices (CSDs), using coherence interconnects (CXLs) for communication between GPUs and CSDs. By leveraging the statistical properties of embedding access patterns, we develop an optimized CSD memory hierarchy and caching strategy. A lookahead embedding scheme enables concurrent execution of embeddings and other operations, hiding latency and reducing memory bandwidth requirements. We evaluate NDRec using real-world and synthetic benchmarks. Results demonstrate NDRec achieves up to 4.33× and 3.97× speedups over heterogeneous CPU-GPU platforms and GPU caching, respectively. NDRec also reduces per-iteration energy consumption by up to 54.9%.**

***Index Terms*—Computational storage, recommendation system, near storage computing, coherence interconnect (CXL).**

I. Introduction

RECOMMENDATION system is an essential building block in many web services such as social networks [38], search engines [6], and e-businesses [5]. Various machine learning methods [21], [22], [43] have been incorporated into modern recommendation systems. With the continuously growing scale and complexity of recommendation tasks, deep neural network (DNN) based recommendation models [11], [30], [34] have received increasing attention for their superior performance; recommendation systems have also become the dominant workload in data centers [16].

The DNN-based recommendation models have a unique architecture that poses some challenges to the training process. One of these challenges is the data-intensive embedding layer, which transforms the categorical features into continuous vectors by looking up and aggregating the parameters from many embedding tables (EMBs). Each EMB corresponds to one input feature and can have a different size, with the largest one having up to billions of entries (e.g., for social media pages). However, only a few entries are fetched in each query [29], resulting in a sparse access pattern that does not benefit from the cache in the CPU and GPU memory hierarchies. This leads to a high memory bandwidth requirement [16] and a low GPU utilization. Moreover, the embedding layer requires a huge memory capacity (up to 10TBs [44]), which grows much faster than the hardware capability. As shown in [32], the memory capacity and bandwidth demand of the DNN-based recommendation models increased by over 16× and 30× in the past five years, while the memory capacity and bandwidth of top-level GPUs only improved by 6× and 2.3×, respectively.

The efficient training of large-scale recommendation models on GPU requires different strategies to deal with the data-intensive embedding layer. One strategy is to shard the embedding tables across multiple GPUs and use all-to-all communication to aggregate the embedding results, as proposed in the original paper of DLRM [30] and later improved by [29]. However, this strategy assumes that the total GPU memory can accommodate the whole embedding table, which is not feasible for certain large-scale models (e.g., with 12TB of parameters, 128 A100 GPUs are needed). Another strategy is to use host memory as backing storage and GPU memory as cache, and move the entries that are needed within a lookahead window to the GPU memory, as proposed by [3], [7], [24]. This solution avoids sharding but still relies on DRAM to store the embedding table, which is costly and could increase the operational expense with the growing size of the recommendation models.

One possible solution to cope with the increasing memory capacity demand of the embedding layer is to use lower-level storage, such as Solid-State Disk (SSD), in the memory hierarchy. However, this solution faces the challenge of limited

Manuscript received 19 October 2023; revised 17 January 2024; accepted 28 January 2024. Date of publication 15 February 2024; date of current version 9 April 2024. This work was supported in part by the NSF under Grant CNS-2112562 and Grant CNS-1822085, and in part by the membership for the NSF IUCRC for ASIC, including Samsung and so on. Recommended for acceptance by N. S. Kim. *(Corresponding author: Shiyu Li.)*

Shiyu Li, Yitu Wang, Edward Hanson, Hai Li, and Yiran Chen are with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: shiyu.li@duke.edu; yitu.wang@duke.edu; edward.t.hanson@duke.edu; hai.li@duke.edu; yiran.chen@duke.edu).

Andrew Chang and Yang Seok Ki are with the Memory Solution Lab, Samsung Semiconductor Inc., San Jose, CA 95134 USA (e-mail: andrew.c1@samsung.com; yangseok.ki@samsung.com).

Digital Object Identifier 10.1109/TC.2024.3365939

bandwidth, both from the SSD and the system interconnects, which cannot meet the data movement requirements of the training process. We propose to overcome this challenge by offloading the embedding layer to the storage side and only transferring the output of the embedding layer to the GPUs. We observe that the computation cost of the embedding layer is mainly determined by factors unrelated to the total volume of the parameters, such as the pooling size, the size of embedding dimensions, and the number of EMBs [29]. This implies that the embedding layer can be computed using a less-powerful device, such as FPGA-based computational storage devices (CSDs), e.g., SmartSSD [25]. FPGA-based CSDs allow us to build customized hardware kernels and exploit the access patterns of the embedding tables. Offloading also enables the further scale-up of the recommendation models, as storing data in storage devices is usually much cheaper than in DRAM or high-end GPUs.

Based on the above observations, we present NDRec – a near-data processing system for training large-scale DNN-based recommendation models in this work. We use CSDs to offload memory-intensive embedding operations, free up GPU resources for compute-intensive operations, and enable full data-parallel training on GPUs. To achieve this, we decompose the embedding layer into two stages: 1) lookup with the old weights and 2) update with the generated gradients. This decomposition allows us to perform 'lookahead' before the weights are updated and thus enable concurrent execution of the embedding layer and the other layers of the recommendation models. Moreover, we design a software-managed cache in the CSD's DRAM based on the embedding tables' access pattern. We evaluate NDRec with both real-world and synthetic benchmarks and show that it can achieve significant speedup and energy saving over existing platforms. Specifically, NDRec can achieve up to 4.33× and 3.97× speedup on real-world benchmarks over a unified-memory-based CPU-GPU platform and GPU caching, respectively. For the synthetic embedding-dominant benchmark, NDRec can achieve over 26× speedup over the GPU baseline. NDRec can also reduce the energy consumption of single training iteration by up to 54.9%.

We summarize our contribution as follows.

- We identify the opportunities of offloading the embedding layer to CSDs and design **a near data processing system, namely, NDRec**, to tackle the memory capacity bottleneck of large-scale DLRM training.
- We propose a **lookahead embedding** scheme that enables the concurrent execution of the compute-intensive layers and the data-intensive embedding layer during training. The concurrency relaxes the bandwidth requirement of the embedding operation.
- We propose a **software-managed caching strategy** on CSDs to hide the long access latency of the SSD and fully utilize the SSD bandwidth.
- We design an **FPGA kernel** on the SmartSSD device for processing the stored embedding table.

The remainder of this paper is organized as follows. Section II presents the background information; Section III analyzes the challenges and opportunities that we found to build the NDRec system; Section IV presents the design of the NDRec system; Section V provides details on the CSD side design; Section VI demonstrates our evaluation results and the corresponding discussions; Section VII summarizes related works, and Section VIII concludes this work.

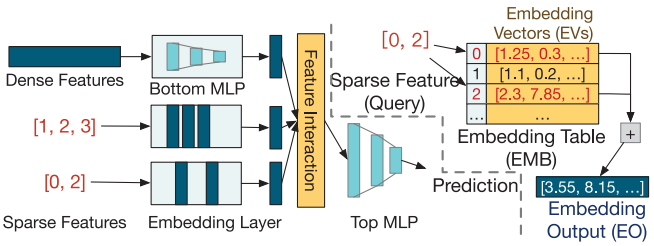

Fig. 1. Left: The general architecture of DLRM. Right: An example of the embedding operation.

II. Background

A. DNN-Based Recommendation Model

We focus on DLRM [30], a representative DNN-based recommendation model that has been widely used in the industry. Fig. 1 shows the general model architecture of DLRM, which takes both dense and sparse features as inputs. Dense features are numerical values, such as length of time. Sparse features are categorical values, such as genres of movies. Dense features are processed by a bottom MLP, while sparse features are transformed into dense vectors by an embedding layer. The outputs of the bottom MLP and the embedding layer are combined by a feature interaction operation and fed to a top MLP for generating the prediction output. The inference process of DLRM handles single or small batches of dynamic input samples, with low latency as the main optimization goal. The training process of DLRM processes large batches of input samples from a static dataset, with high throughput as the main optimization goal. The training process is typically performed on GPU or other powerful accelerators.

The embedding layer consists of multiple EMBs, each corresponding to one sparse feature. Each row of an EMB i.e., an embedding vector, represents one category, and each sparse feature contains one or more indices that refer to the rows of the EMB. In the forward pass of the embedding layer, we fetch the *embedding vectors* (EVs) from each EMB according to the indices in the corresponding sparse feature and aggregate these vectors. Most recommendation models use summation as the aggregation operator. To avoid confusion with the embedding vector, we use *embedding output* (EO) to denote the aggregated result from one embedding table. These terms are also illustrated in Fig. 1.

B. Computational Storage

CSDs are introduced to address the I/O bottleneck by allowing the processing of the data in place. Storage vendors have proposed two paradigms of CSDs: (1) integrating the processing units into the SSD controllers [40]; (2) building a separate

processing unit near the SSD with a private P2P link [25]. We select the latter type of CSD design, specifically the SmartSSD, for its flexibility in customizing the hardware. SmartSSD builds a separate FPGA chip alongside the SSD device and provides a private PCIe 3.0 ×4 link between the two parts. The FPGA has a dedicated 4GB DRAM for a customized accelerator design. The functionalities related to SSD—interface handling, wear leveling, error correction, etc.—are implemented in the SSD controller. The data movement between SSD and FPGA's DRAM can be invoked with a command by the host program, but the actual transfer bypasses the host [41].

C. Coherent Interconnect

Conventional system interconnects, for example, PCI-express (PCIe), were designed for IO devices. Although such interconnects are able to provide high bandwidth, they are inefficient for fine-grained transfers. Moreover, handling the coherence and synchronization with the software will incur extra overhead. Novel coherent system interconnects, e.g., Compute eXpress Link (CXL) [37], are proposed to address this issue. CXL utilizes the physical layer of PCIe and provides *CXL.io*, *CXL.mem*, and *CXL.cache* protocols for devices. *CXL.io* protocol is an enhancement of the PCIe protocol, while the *CXL.cache* and *CXL.mem* protocols have smaller packetization overhead and a higher priority, which creates low access latency. Three types of devices are defined in the CXL specification. The type-1 device only has *CXL.cache* protocol and hides its device-attached memory. Type-2 is the accelerator device with both *CXL.cache* and *CXL.mem* protocols. Type-3 devices have *CXL.mem* protocol and can be used as a memory expansion of the system.

Type-2 and Type-3 devices can (selectively) map their attached memory (e.g., the GPU memory) as host-managed device memory (HDM). HDM is mapped to the CPU's host physical address (HPA) space and is directly accessible to the host as cacheable memory. The coherency of HDM can either be managed by the host (HDM-H) or the device (HDM-D or HDM-DB). HDM-DB utilizes back invalidation channels to enable direct snooping by the device to the host. Device-coherent HDM has a bias-based coherency model. The cacheline can be host-biased or device-biased, whose coherency is resolved at the root complex or the device coherence engine. The bias mode can be either explicitly controlled by software or managed by hardware. The device-biased mode saves unnecessary traffic to/from the host for internal memory access in Type-2 devices. Type-2 devices are also able to access any cacheable memory within the system address space with hardware coherence. We adopt the latest CXL 3.0 standard [13] in our design since it supports multiple Type-2 devices per root port, back invalidation-based device coherent HDM, and multi-level switching.

D. CXL-Enabled SSD

The integration of CXL introduces an enticing prospect for leveraging SSDs as an extension of the main memory. The substantial storage capacity and cost-effectiveness of SSDs offer a promising solution to address the challenges posed by memory-intensive applications. A prior proposal [17] outlines a practical approach to developing CXL-enabled SSD devices by enhancing the SSD controller. For the SSD to function as a Type-3 device, it must provide byte addressability and, at a minimum, support access at the cacheline granularity. This requirement contrasts with the page-grained access inherent in NAND flash, potentially leading to significant read/write amplification issues. Additionally, the extended access latency and finite endurance of SSDs pose formidable obstacles in constructing such devices. A recent investigation [42] delves into design options aimed at overcoming these challenges. The study demonstrates that, through the implementation of caching, prefetching, and software assistance, it is feasible to construct CXL-enabled SSDs with commendable performance. Notably, industry vendors are actively exploring CXL-enabled SSDs and have showcased prototypes of such designs [12].

III. Motivation

DNN-based recommendation models have increasing memory capacity and bandwidth requirements, but their computation demand does not scale proportionally. Based on the representative model configurations from Meta [29], we observe that: 1). The computation demand of the embedding layer is determined by the embedding dimensions (i.e., the number of EVs to reduce) and the number of tables, not by the number of parameters. 2). The models with more computation demand in the embedding layer (i.e., with more embedding tables) also have more and wider MLP layers, which are the dominant operators. These observations suggest that the embedding layer can be offloaded from GPU without creating a new bottleneck.

SmartSSD is a potential candidate for offloading the embedding, as it provides large and cheap capacity with SSD. However, the limited memory bandwidth and long access latency of SSD pose challenges for implementing embedding. We explore opportunities in the overall training process to overcome these challenges.

Opportunity 1: Use the knowledge of access pattern to embedding table for scheduling. Unlike in inference, we can know the exact inputs for future iterations during training. This information can help us schedule the data movement among SSD, DRAM Cache, and on-chip buffer. Furthermore, previous works [32], [35] have shown that the embedding vector access distribution is skewed. A small subset of embedding vectors accounts for a large fraction of the total accesses. We can leverage this statistical feature in the caching strategy design to minimize access to SSD.

Opportunity 2: Relax the dependency to reduce the bandwidth demand. For each training iteration, the embedding operation has a read-after-write (RAW) dependency with the weight update process of the previous iteration. The subsequent forward process also depends on the embedding operation (RAW). This means that even if we offload the embedding operation to other devices, we still have to wait for the previous training iteration to finish before the embedding operation can start and produce the result for GPU. The embedding operation cannot run in parallel with other processes on GPUs. These

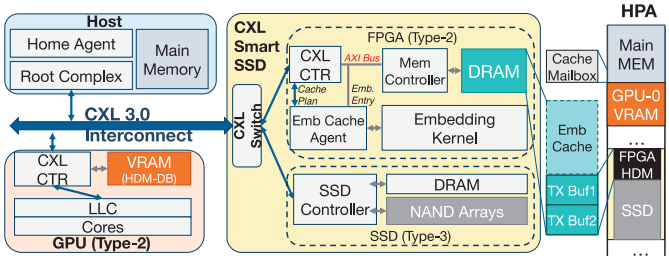

Fig. 2. The architecture of the NDRec system and the allocation of the host physical address space (HPA). The blue arrow indicates coherent traffic in the CXL subsystem while the gray arrows show internal traffic. 'HDM-DB' stands for host-managed memory with device coherent using back-invalidation.

dependencies drive the high memory bandwidth demand. We found it is possible to break the embedding operation into two stages. In the first stage, we use the old embedding vectors to compute an intermediate embedding output. This stage is memory-intensive but can start earlier in the previous iteration. In the second stage, we correct the intermediate result when the gradients are available. This relaxation preserves the 'illusion' of respecting RAW dependencies (i.e., the optimizer behavior is unchanged) and enables concurrent execution between GPU and SmartSSD. The concurrent execution can hide the latency of the embedding operation, thus lowering the bandwidth requirement for the same target training throughput.

Opportunity 3: Trade the cheap GPU computation for expensive memory bandwidth. The aforementioned correction to the embedding output can be represented as a matrix multiplication between the gradients and the indicator matrix of the overlapped embedding vectors between two iterations. The batched matrix multiplication is a relatively simple and well-optimized task for GPU. Since the cost of a compute-saturated task is lower than the memory-saturated one [4], we argue that it is worthwhile to trade the extra computation on GPU for significantly reducing the bandwidth demand of embedding operation on SmartSSD. Moreover, the extra matrix multiplication on GPU has a negligible overhead compared with the MLPs.

IV. NDREC SYSTEM

A. System Architecture

The proposed design is shown in Fig. 2. NDRec system consists of host processors with CXL root complex, CXL-enabled GPUs, and CXL-enabled SmartSSDs. A CXL-enabled GPU is a Type-2 device that supports io, cache, and memory protocols. Meanwhile, a CXL controller replaces the original PCIe controller and handles the protocol traffic on the GPU. The CXL controller has a device coherence engine that resolves accesses from the CXL.mem protocol or peer caches and forwards cache coherence messages to the last-level cache. GPU memory (VRAM in the figure) is mapped as HDM-DB, which uses back invalidation channels to maintain coherency with the host. This is because the access latency from GPU's LLC to GPU memory is critical for the performance of the kernels running on GPU. We assume that hardware manages the bias mode autonomously, as suggested by the spec [13].

Because there is yet to be any CXL-enabled SmartSSD on the market, we extrapolate its design based on existing architecture. The device consists of a CXL switch, an SSD with CXL interface as a Type-3 device, and an FPGA as a Type-2 device. For the SSD, we follow a previous proposal [17] where the SSD controller directly supports memory protocol and byte addressability. We use non-deterministic (ND) and bufferable (BF) attributes for all requests to the SSD for performance reasons, i.e., improving asynchrony and overlap of compute and multiple in-flight requests. The original controller functions, such as wear leveling, error corrections, etc., remain unchanged. Load/store request from the memory protocol is translated into read and write requests and sent to the controller. The control and management commands can be issued through the io protocol. From the system perspective, the SSD can now be accessed with load/store requests as regular memory. For the FPGA, we replace the original PCIe endpoint with a CXL controller, similar to the CXL-enabled GPU. The CXL controller also has a coherence engine for resolving the request to the FPGA memory. We map part of the FPGA memory as HDM-DB, which will be used as transaction buffers (Section IV-E). Software controls the bias mode of this region. The coherence engine handles the external and internal access to the transaction buffers to maintain coherency. The rest of the FPGA memory is used as the embedding cache region and reserved as private device-attached memory (PDM).

Remaining resources of the FPGA are reserved for customized logic where we build an embedding cache agent and an embedding kernel. The embedding cache agent cooperates with the host cache planning process to manage the embedding cache (Section V-A). A hardware kernel performs the forward and backward computation of the embedding operation (Section V-B). Meanwhile, the embedding kernel sends read/write requests of the EVs to the embedding cache agent, in which the counter and statistics of the EV are updated, and the request is fulfilled by forwarding the request to the AXI bus. Accesses to EVs are non-coherent since EVs reside in the reserved PDM region (embedding cache region). Meanwhile, the embedding cache agent pulls cache plans from the cache mailbox region in the host memory and executes these plans by sending read and write requests to the CXL controller. The EVs in the embedding cache can be viewed as coherent snapshots of the EVs stored in SSD. Load/eviction operations are implemented with memory copy.

All aforementioned memories, except for the embedding cache region and the SSD's DRAM, are mapped into host physical address space. The SSD controller uses the DRAM for running the firmware and buffering data and is transparent to the system. The system can have multiple GPUs and/or SmartSSDs, which can be connected to the root port directly or through multiple levels of switching. In this work, we study the configuration with up to two levels of switching and a simple tree topology. We avoid using more advanced fabric management features in CXL 3.0 due to a lack of performance studies to cross-validate against.

B. Lookahead Embedding

As we mentioned in Section III, the RAW dependencies of the embedding layer prevent concurrent execution without

speculation. This restriction imposes a tight latency constraint on the memory-intensive embedding operation, thus requiring high memory bandwidth. By offloading the embedding operation, we also hope to perform the embedding operation concurrently with other operations running on GPU and relax the bandwidth requirement.

Assuming the recommendation model uses the summation as the pooling operation, the computation of the embedding output can be represented as,

$$v_{i,j}^{t} = \sum_{k=1}^{N_{i,j}^{t}} E_{j}^{t}[O_{i,j}[k]], \tag{1}$$

where the subscripts i, j, and t represent the j-th sparse feature of the i-th sample in the t-th iteration; E_j is the j-th EMB, $O_{i,j}$ is the array of indices of the current query, and $N_{i,j}$ is the number of those indices. The backward process is done by aggregating the gradient vectors from the samples that include the EV in the forward pass,

$$\frac{\partial L}{\partial E_{j}^{t}[k]} = \sum_{i} \frac{\partial L}{\partial v_{i,j}^{t}} \mathbb{I}_{k \in O_{i,j}^{t}}, \tag{2}$$

where $I_{k \in O_{i,j}^{t}}$ is an indicator function.

Thus, we may decompose the process of computing EVs in Equation (1) into two stages, embedding lookup with the old embedding table and correction with the new gradients. Specifically, if with SGD optimizer, the EV in equation (1) can be calculated as,

$$v_{i,j}^{t} = \sum_{k=1}^{N_{i,j}^{t}} E_{j}^{t-1}[O_{i,j}^{t}[k]] - \eta \sum_{p \in O_{i,j}^{t}} \sum_{q}^{N_{batch}} \frac{\partial L}{\partial v_{q,j}^{t-1}} \mathbb{I}_{p \in O_{q,j}^{t-1}}, \tag{3}$$

where η is the learning rate, N_{batch} is the batch size, p covers all indices related to the i-th sample of the current iteration, and q captures the sample indices that include the entry p in the previous iteration. The second term indicates the difference between the embedding vector calculated with the old embedding table and the updated embedding table. It can also be expressed in the vector form,

$$M_i \frac{\partial L}{\partial v_{:,j}^{t-1}}, M_i = [\sum_{p} \mathbb{I}_{k \in O_{1,j}^{t-1}}, ..., \sum_{p} \mathbb{I}_{k \in O_{N_{batch},j}^{t-1}}]^{T}. \tag{4}$$

Since the indices of each batch is determined for each training run, the vector M_i can be generated through the preprocessing on CPU. Thus, we can compute the first term in Equation (3) on SmartSSD concurrently with the training process running on the GPU. The result is sent to GPU. Then, the correct EV can be obtained by performing a matrix multiplication on GPU according to the Equation (3).

We name this scheme as *lookahead embedding*. Fig. 3 depicts the lookahead embedding process. Lookahead embedding enables the concurrent execution of GPU and the SmartSSD. Fig. 4 compares the training timelines with and without the lookahead embedding scheme. For example, when the GPU is processing the forward/backward for the n-th iteration, the backward process updates the embedding vectors with the

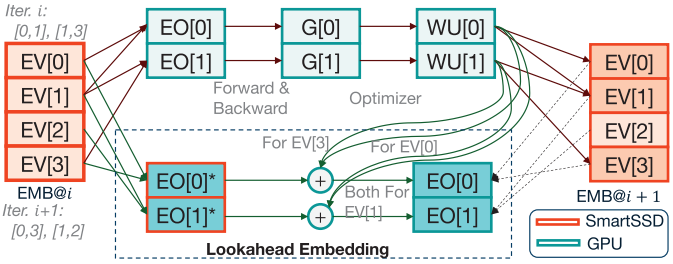

Fig. 3. The illustration of lookahead embedding. There are two samples per batch. 'G' and 'WU' denote gradients and weight updates, respectively. 'EMB@i' stands for the embedding table at the start of the i-th iteration. We omit the lookahead embedding for the i-th iteration for readability.

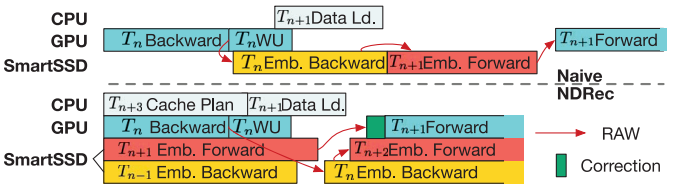

Fig. 4. The illustration of the training timeline of naïve offloading and NDRec. 'WU' stands for weight update. The embedding backward for the $(n-1)$-th iteration can be overlapped with the embedding forward of the $(n+1)$-th iteration by forwarding the updated EVs.

gradients generated at the end of the $(n-1)$-th iteration. Meanwhile, the forward process speculatively calculates the embedding output for the $(n+1)$-th iteration with the forwarded embedding vectors updated by the backward process of the $(n-1)$-th iteration. We will discuss the forwarding in Section V-B. In other words, the lookahead embedding removes the RAW dependency between the forward process of T_{n+1} and the backward process of T_n. Instead, the forward process of T_{n+1} relies on the output of the backward process of T_{n-1}.

Although we only discussed the model with summation as the pooling operator, as previous works did [19], [20], [23], [32], the proposed lookahead embedding also applies to other linear operators (e.g., mean, weighted sum, etc.). We also noticed sum and mean are the only pooling operators supported by a majority of recommendation model frameworks [31], [36] or adopted by the published model configurations [1], [29]. The decomposition made by lookahead embedding does not make any approximation in embedding lookup. The EOs used by forward process are the accurate one computed in two stages. The EV update is delayed by one iteration but still follows the original computation formula. Thus, lookahead embedding does not alter the behavior of the training process and has no impact to the accuracy.

C. Task Coordination

The major objective of NDRec is to offload the embedding operation to computational storage devices and enjoy the large cheap capacity provided by SSD. Unlike the original sequential execution flow of the model, offloading decomposes the recommendation model training into multiple smaller tasks that can

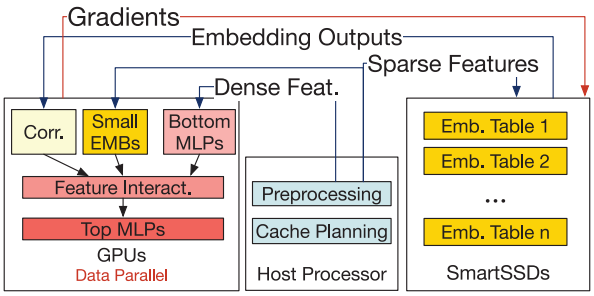

Fig. 5. The task coordination in NDRec system.

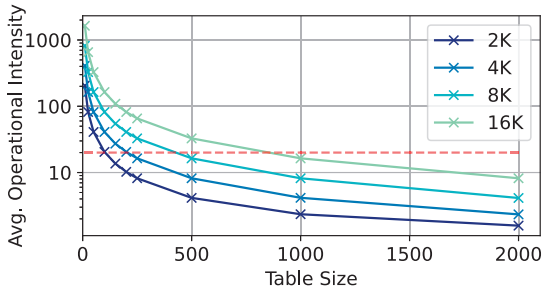

Fig. 6. The operational intensity (FLOPS/DRAM byte) with different table sizes and batch sizes. The dashed red line indicates the operational intensity corresponding to the intersection of bandwidth ceiling and peak performance ceiling on the roofline model of RTX A5000.

run concurrently. Apart from the computation of MLP layers remaining in GPU and the embedding layer offloaded to the SmartSSDs, we also need to run the preprocessing process to feed the input features and one or more cache planning processes (depending on the number of the SmartSSD devices) to schedule the data movement.

We illustrate the task coordination of the NDRec system in Fig. 5. The GPUs process the compute-intensive part of the recommendation model (i.e., MLPs) with the data parallel strategy. Apart from the MLP layers, we also place small embedding tables on the GPU devices. The small embedding tables show the higher operational intensity (if we assume the embedding vectors can fit into the L2 cache) according to the pigeonhole principle, i.e., if the total number of embedding vectors is much smaller than the number of accesses per iteration, some embedding vectors must be accessed multiple times. The parameters of the rest of the embedding tables are stored in the SSD, and the FPGA handles the computation of the embedding lookup in each SmartSSD device. The bandwidth requirement determines the placement of embedding tables. We will provide quantitative analysis on the embedding table placement in Section IV-D.

The FPGA DRAM of each SmartSSD device is split into multiple regions, as shown in Fig. 2. Two regions work as transaction (TX) buffers to hold the EOs of the current iteration (old EO buffer) or store the result of the lookahead embedding (new EO buffer) for the next iteration. The total size of embedding outputs determines the size of the TX buffer region. If we assume TX buffer 1 is used as the old EO buffer and TX buffer 2 is used as the new EO buffer, GPU will fetch from the TX Buffer 1, while the embedding kernel will write the result of lookahead embedding into the TX Buffer 2. The two buffer will switch their roles at the beginning of each iteration. At the beginning of the next iteration, TX buffer 1 becomes the new EO buffer, and the newly generated EOs will overwrite the data. TX buffer 2 becomes the old EO buffer and keeps its data for GPU to fetch. The embedding cache region occupies the remaining capacity of the memory. Currently, we did not further partition the embedding table and assign them to different SSDs since the cross-device access incurs high overhead.

During one training iteration, the data loader and preprocessing process load the raw input data from storage or main memory. The dense features and sparse features for small embedding tables are sent to the GPU, while the sparse features for large embedding tables are sent to the SmartSSDs. The forward/backward computation of MLP layers on GPU and forward/backward computation of the embedding layers on SmartSSDs run concurrently, as shown in Fig. 4. Apart from the computations, we have one cache planning process running on the CPU for each SmartSSD. The cache planning process maintains the metadata of the entries in the embedding cache, schedules the data movement, and writes the movement schedule to the cache mailbox. Before starting a new iteration, the system will perform a synchronization to guarantee all queuing data movement commands have been committed. We will discuss synchronization details in Section IV-E.

D. Embedding Table Placement

GPU or SSD. As aforementioned, the small embedding tables typically show higher operational intensity (FLOPs/DRAM Byte), which indicates assigning them to GPU is a better solution. We adopt a simple rule in determining the threshold on the placement decision–the operational intensity should be large enough to avoid the throughput being bounded by the memory bandwidth. We proceed to compute the average operational intensity (AOI), where AOI is defined as the ratio of the number of floating-point operations (FLOPs) needed for embedding lookup per batch to the table size. This metric serves as a proxy for estimating the position on the roofline analysis, helping discern whether the embedding lookup for a particular table is constrained by memory bandwidth or computation. Through a comparative analysis with the GPU roofline, we employ this heuristic to ascertain the optimal placement. The analysis results on our experiment platform are shown in Fig. 6. We use this heuristic for the placement of the embedding table in the following experiments.

Among SSD. The key metric for determining the table placement across SSDs is the achieved aggregated bandwidth. As mentioned in [32], the bandwidth requirement of EMBs is mainly related to the pooling size, i.e., how many embedding vectors are involved in each embedding lookup. Since we focus on maximizing the bandwidth between DRAM and FPGA and most of the embedding vectors can stay on the on-chip buffer within one training iteration, we use the number of unique embedding vectors per iteration rather than the pooling size as the proxy of the bandwidth requirement. Thus, to derive an

EMB placement plan, we sample 1% of the training data (as suggested in [32]) and calculate the average number of unique EVs per iteration. Then, we sort this number and assign a pair of tables from both ends of the sorted list to the SSDs in a round-robin fashion.

E. Communication

NDRec system performs a synchronization at the end of each iteration to guarantee resolution of the five key operations or transactions. Specifically: 1) the first stage of the lookahead embedding has finished on SmartSSD, and the result has been written into one of the TX buffers; 2) the backward process on GPU has finished, and gradients of embedding tables have been produced; 3) the backward process on FPGA has finished, and all updated are committed to the FPGA DRAM; 4) the embedding cache agent has committed all data movement schedules; 5) the cache planning process running on CPU has written the new data movement schedule to the mailbox and sent the pointer of the mailbox to the embedding cache agent on FPGA. Afterwards, a new iteration for training can commence.

At the beginning of each iteration, a copy of the EMB gradients is created on the GPU, and the corresponding address is passed to the embedding kernel. Then, we issue the command to FPGA and swap the role of the TX buffers. After swapping, the old EO buffer (which contains the EOs for this iteration) is configured as host-biased and ready to be fetched by the GPU, while the new EO buffer (which contains stale EOs) is configured as device-biased and ready to receive the new result generated by the embedding kernel. The write to the new EO buffer will issue a back invalidation request, flushing the cache line in all peer caches (i.e., GPU caches), and avoiding the stale data used by GPU in the future. GPU then fetches the result from the TX buffer and performs the training process, while the embedding kernel loads the gradients directly from GPU memory to start the backward process. The embedding cache agent pulls the data movement schedule from the mailbox region in host memory and executes the schedule to write the EVs in DRAM back to SSD and load new EVs. Since the embedding cache is software-managed, we need to perform a memory copy rather than directly issue a cacheable load request to avoid data consistency issues. The memory copy also helps avoid the impact of unpredictable SSD access latency during forward/backward.

V. Near-Storage Embedding

A. Embedding Cache Design

As discussed in previous works [32], [35], accesses to embedding tables show a power-law distribution. Thus, we can build an embedding cache to capture this access pattern, reuse the frequently used embedding vectors, and reduce the expensive SSD access. Previous designs [3], [7] utilize the information of upcoming iterations to build a software-managed lookahead cache and preload the necessary embedding vectors to the cache. On the CPU-GPU platform, the cache management process running on the CPU needs to not only *plan* for the data movement but also *collect* and *exchange* these data from the main memory and GPU memory. Both cDLRM [7] and ScratchPipe [24] show the overhead of these processes running on the CPU could dominate the latency, especially when we have multiple caches to manage (i.e., with multiple GPUs). For SmartSSD, only the *plan* phase needs to be done by the CPU. The data movement is then handled by the embedding cache agent on SmartSSD, runs asynchronously with the computational kernel, and does not require the involvement of the CPU. Thus, we can overlap the planning and exchange phases and have a higher tolerance for the search cost.

We built an embedding cache in the FPGA DRAM on each SmartSSD. The cache design is based on two principles: 1) trade the manipulation of metadata for the movement of embedding vectors; 2) Use the access pattern's statistical feature to reduce SSD access. Moreover, unlike traditional cache, we need to prefetch all necessary embedding vectors *before* the iteration and guarantee the hits during the forward and backward process of the embedding. Thus, we design the embedding cache as a software-managed fully-associated cache, with each cache block representing an embedding vector. The cache blocks are stored in FPGA DRAM, while all metadata is stored in the main memory and manipulated by the cache planning process on the CPU. The metadata is maintained as a table of (EV ID, Address) pairs. We also maintain an access counter for each embedding vector presented in the cache to evaluate the access frequency of the EV. Apart from the main table that contains the EV-address pairs, we also have a hot-entry table and an in-use table to utilize the statistical information of the EV access. The main table, hot-entry table, and in-use table are exclusive to each other, i.e., each EV can only be present in one of these three tables. The hot-entry table contains frequently accessed EVs. The EV will be promoted to the hot-entry table if the counter surpasses a certain threshold. The EVs in the hot-entry table will not be evicted. The in-use table tracks EVs used by the process running on the SmartSSD (i.e., four sections for T_{n-1}, T_n, T_{n+1}, and T_{n+2}). The entry in the in-use table has an extra reuse bit indicating whether they will be used by the subsequent iterations.

We illustrate the embedding cache in Fig. 7. The cache planning process, which runs on the CPU, generates a data movement plan and writes it to the mailbox region in the main memory. The mailbox is implemented as a circular buffer, where the plan for the next iteration is appended to the end of the buffer. Upon reading the plan from the mailbox, the embedding cache agent on the FPGA executes the plan. If we assume GPU is running forward and backward processes for the n-th iteration, as shown in Fig. 4, the forward process on SmartSSD computes for the $(n+1)$-th iteration. Meanwhile, the embedding cache agent executes the data movement plan to collect the EVs required by the $(n+2)$-th iteration, while the cache planning process on the CPU generates the plan for the $(n+3)$-th iteration. The cache planning process will run two iterations ahead of the current forward process running on the SmartSSD so that data movement for the next iteration can overlap with cache planning.

In the cache planning phase, we first create a section in the in-use table to collect all EVs to be used by the next iteration

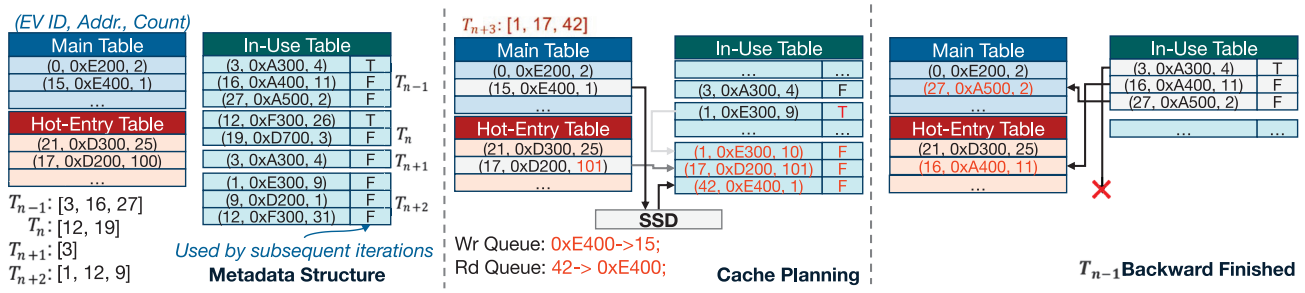

Fig. 7. An example of the metadata structure and the management process of the software-managed cache. The changes in each step are highlighted. This figure shows only the manipulation of the metadata and the data movement is executed only when the queues are submitted.

(i.e., T_{n+3}). The metadata of these EVs can either be from the in-use table, that main or hot-entry table, or a new entry created for a load from SSD. If the EV is also used in the previous iterations (e.g., EV 1 in the figure), we directly get the metadata from the corresponding section in the in-use table and update the reuse bit, indicating the EV should not be returned to the main or hot-entry table. For the rest of the EVs, we first search in the hot-entry table, then the main table for the entry. The hit will remove the EV from the main or hot-entry table and add it to the new section of the in-use table. If the EV cannot be found in both tables (e.g., EV 42 in the figure), we generate an SSD access command and randomly select one EV (e.g., EV 15 in the figure) in the main table for eviction. A new entry will be created in the in-use table for this EV. The SSD access and eviction commands are sent to two priority queues, where a coalescer will check whether the command can be merged to improve the transfer efficiency. Finally, at the synchronization point, the plan generated from the two queues will be written into the mailbox. The plan guarantees that the write command (i.e., eviction) will be first executed to avoid the dirty entries being overlapped by the new entries. Meanwhile, we will also flush the table for T_{n-1} by returning the EVs not used by later iterations back to the main table.

When there is a hit, the counter related to the corresponding EV will be updated. We will compare the updated counter with a predefined threshold and promote the entry to the hot-entry table when the entry is flushed from the in-use table if it is larger than the threshold. When the hot-entry table is full (i.e., larger than a predefined size), the tail entry will be evicted back to the main table from the hot-entry table. We will discuss the selection of the parameters for hot-entry table in Section VI-C.

B. Embedding Kernel

The embedding kernel design aims to maximize the reuse of the embedding vectors presented on-chip and reduce off-chip access. We utilize the overlap of embedding vectors between consecutive iterations to achieve this goal. If we take the timeline presented in Fig. 4 for example, when we start the T_{n+2} forward process, the EVs used by the forward process of T_{n+1} and not used by the backward process of T_n are presented in the on-chip buffer and does not need to be reloaded from the embedding cache. Similarly, the backward process of T_n can reuse the EVs from the backward process of T_{n-1}.

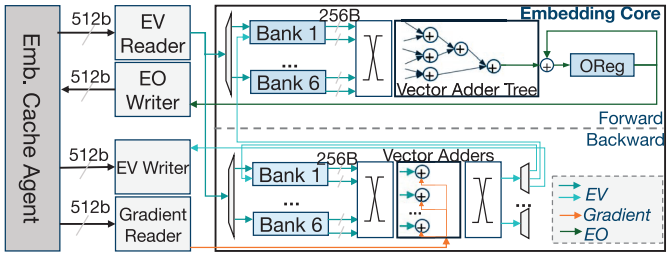

Fig. 8. An example of the embedding kernel configuration with embedding dimension of 64. The control signals and the write ports on Bank 6 of both buffers are omitted for readability.

Meanwhile, the EVs updated by the backward process of T_n can be forwarded to the forward process of T_{n+2} to run these two processes concurrently.

The architecture of the proposed embedding kernel is depicted in Fig. 8. We used a banked buffer design where each buffer bank corresponds to the number of dual-port BRAM blocks (32-bit per port [9]) that can provide the width of two embedding vectors. The bank size depends on the embedding dimension. For example, if the dimension is 64, each buffer bank of the embedding table contains 64 BRAM blocks or 32 URAM blocks. We can build 12 such buffer banks given the available memory resources for the kernel and memory controller on SmartSSD [33]. We then evenly split the buffer banks and used them for the forward and backward process, respectively. For the forward process, the EVs read from the buffer are then sent to a crossbar and a vector adders tree to generate the partial embedding output. The output is then accumulated in the output register and sent to the result writer when all EVs are reduced. For the backward process, one gradient vector (processed by the optimizer) is loaded from the gradient reader per cycle. All EVs related to the gradient vector are read from the buffer, updated, and then written back to the buffer. When all gradients related to the EV have been applied, the updated EV will be forwarded to the buffer for the forward computation or written back to memory if it is not used in subsequent iterations. The assignment of the EVs to buffer banks is done during the preprocessing phase. We assign the EVs in a round-robin manner and swap the assignment with other EVs in the same iteration if there is a bank conflict. When the forward/backward process is done for one iteration, the buffer will hold EVs for the next iteration to reuse them.

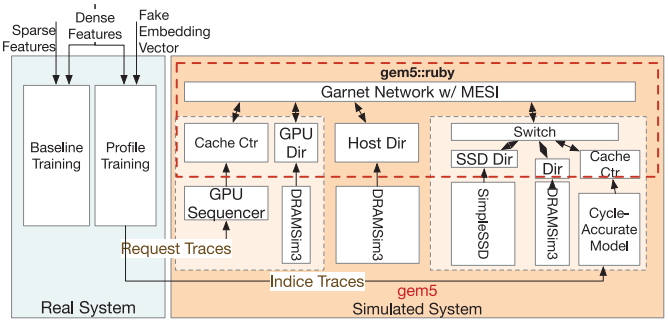

Fig. 9. The simulation framework.

TABLE I
HARDWARE SETUP FOR THE EVALUATION

Server	
CPU	AMD EPYC 7352×2 @ 2.8GHz
GPU	NVIDIA RTX A5000 24GB ×4
DRAM	16 × DDR4 3200 64GB DIMM
Mem Ctrl.	8 Channels/Node × 2 Nodes
Simulation	
DRAM	DDR4 2400 8Gb ×4
SSD	Samsung 983 DCT 3.84 TB
Interconnect	CXL 3.0x16@16GT/s/lane
Kernel Freq.	250MHz
# of SmartSSDs	4

The kernel is connected to the memory controller through the AXI bus. We build two reader units and two writer units. The EV reader receives the requests from the controller and initiates the request to the memory bus. Since the length of each EV could be larger than the width of the memory port, the reader will send a burst request, compose the response data into one vector, and send it back to the block. The gradient reader sends requests to load the gradient from GPU memory and sends it to the vector adder for the backward process. The EO writer stores the embedding output back to memory. Meanwhile, the EO writer updates the embedding table entries in the embedding cache. The readers/writers are connected to the AXI bus through 512-bit ports to match the width of the memory controller.

VI. EVALUATION

A. Experiment Method

Platform. Due to the lack of commercially available CXL 3.0 systems or official support for P2P access between GPU and SmartSSD, we are not able to fully evaluate NDRec on a physical system. Instead, we design our evaluation setting based on RecNMP [19]. We use TorchRec [36] to run the training pipeline on the real GPU-based system and replace the embedding operation with a predefined tensor. The traces of the training process, containing the detailed latency of each event during training, are then exported using profiling tools. Then, we feed these traces into our simulation framework. The experiment methodology is depicted in Fig. 9. We performed our study with gem5 [28]. The CXL is simulated using the Ruby memory system and Garnet network model [2] with the port, link, and switch latencies shown in [26]. We modified the MESI protocol provided in Ruby to support CXL. We downgrade the link bandwidth to make a fair comparison with the PCIe 4.0 bus in the baseline system. The proposed embedding kernel design is implemented with Xilinx HLS and verified on the SmartSSD device. Then, we built a cycle-accurate simulation model for embedding kernel running on SmartSSD according to the implementation result. DRAMSim3 [27] and MQSim-CXL [42] are used to simulate the DRAM and SSD with a CXL interface in the SmartSSD device, respectively. Table I lists key hardware setups and parameters we used for evaluation. We use 200 as the hot-entry threshold and 2048 as the hot-entry table size in embedding cache. The choice of these parameters will be discussed in Section VI-C. We extract the traces of 100 iterations in the middle of one epoch for simulation and report the average latency.

TABLE II
MODEL CONFIGURATIONS

Model	# MLPs	MLP Size (Mean/Max)	Dim	Tables	EMB Size
Kaggle DAC	7	231(512)	16	26	2.06 GB
Criteo Terabyte	8	465(1024)	128	26	91.10 GB
Random-XL	7	460(1024)	128	52	538.90 GB
Random-MLP	20	1590(5120)	64	16	75.90 GB

Benchmarks. We use two publicly available datasets, Kaggle Display Advertising Challenge (DAC) [18], and Criteo Terabyte [14]. Both datasets have 13 dense features and 26 sparse features. The Kaggle DAC and Criteo Terabyte have 39.3M and 645M training samples, respectively. We also create two synthetic datasets that follow the distribution of Terabyte. We create two models for the random datasets to evaluate the configuration with higher computation demand for embedding (Random-XL, pooling size 40) and the MLPs (Random-MLP, pooling size 10), respectively. The details on model configurations are listed in Table II.

Baselines. We select two optimized out-of-core training implementations provided by the TorchRec framework, UVM and UVM-Caching, and a look-ahead caching solution cDLRM [7] as our baselines. We compare NDRec with a near-memory processing solution based on the design of RecNMP [19]. We follow the design of RecNMP-opt, implement the performance model following the description in the paper, and connect the NMP-enhanced memory modules with the system as a type-3 device in CXL (NMP-CXL). We follow the parameters in the original papers with 2 DIMMs × 4 Ranks configuration and only increase the number of channels to 16 to have sufficient capacity to hold the model. We did not compare with the more recent work RecShard [32] since the publicly available datasets are too small for RecShard, as mentioned by the authors. The randomly generated dataset may make an unfair comparison since RecShard Relies on the statistical features of the embedding tables. The UVM strategy uses the unified virtual memory feature provided by the GPU and stores the embedding tables in DRAM, while UVM-Caching adopts a LRU cache in EV granularity to manage the movement of embedding vectors. We

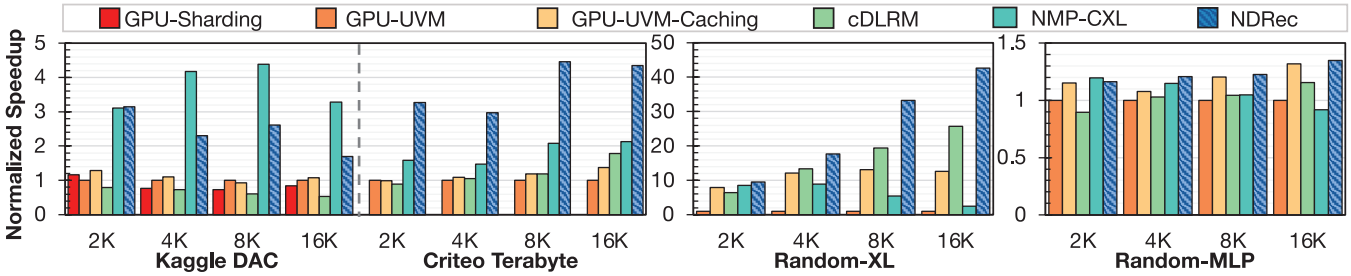

Fig. 10. Normalized speedup of NDRec and baseline approaches over UVM-based GPU training on 4 GPUs.

use the codebase published by the authors [10] with the recommended parameters discussed in the original paper to reproduce the result of cDLRM on our server. For the Kaggle DAC model whose parameters are able to fit into the GPU memory, we also evaluate the naïve table-wise sharding strategy.

B. End-to-End Results

Fig. 10 presents the normalized speedup of NDRec over baseline methods. For the small model, Kaggle DAC, we achieve up to 2.6× speedup over the UVM method. The results also show that the increasing batch size diminishes the advantage of NDRec. This trend originates from the small model size, which makes most embedding vectors reside in GPU memory, even with the simplest UVM strategy in the later iterations. The negligible gap between UVM and caching can also prove this explanation. For the 8K batch size, static (UVM) and look-ahead (cDLRM) caching introduce overhead over UVM and lead to performance degradation. NDRec is slower than NMP-CXL, and the gap deepens as the batch size increases. This is due to the small size of Kaggle DAC. The NMP solution benefits from its high internal bandwidth. As for Criteo Terabyte, we achieve up to 4.33×, 3.97×, and 2.13× speedup over UVM, cDLRM, and NMP-CXL, respectively. A larger batch size helps NDRec explore the reuse of embedding vectors in the embedding cache and on-chip buffer. For the NMP solution, the impact of limited rank cache capacity and the interference of weight updates outweighs the benefit of the larger bandwidth. These two real-world benchmarks demonstrate the advantage of the NDRec system. For small models, even if the whole embedding table can fit into the GPU memory, the concurrent execution of embedding lookup and the rest of the model still achieve noticeable speedup. Furthermore, in a multi-GPU environment, NDRec mitigates the need for additional synchronization resulting from all-to-all communications, a common challenge associated with sharding. The overhead of synchronization can be particularly significant in the training of smaller models. For large models, the embedding table exceeds the size of GPU memory. NDRec can eliminate the frequent data movement between main memory and GPU through the system interconnect with limited bandwidth.

The two synthetic benchmarks, Random-XL and Random-MLP, illustrate two extreme cases where the computation of embedding or MLP dominates the training time. When the embedding layer dominates the overall cost, all strategies

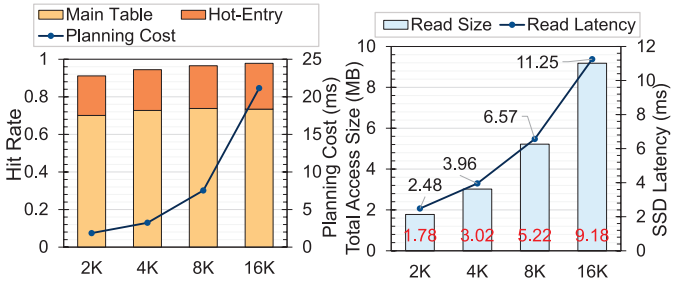

Fig. 11. Left: Hit rate and cache planning overhead; right: average data movement size and latency to SSD per iteration.

except for the NMP-CXL show over 8× speedup over the UVM baseline. NDRec further shows up to 3.38× and 1.71× speedup over UVM caching and cDLRM, respectively. The increased number of embedding tables and a larger pooling size contribute to the opportunities for exploring embedding vector reuse. We also notice the speedup of NMP solution decreases as the batch size increases. This is due to the limited rank cache size in NMP solution causing a low hit rate. The Random-MLP, however, achieves up to 1.35× speedup with different approaches. Since the MLP computation is the dominating factor in the training latency, the data movement cost can be hidden by the computation. These two synthetic benchmarks prove the effectiveness of NDRec in addressing the overhead brought by the embedding operation. Since the synthetic benchmarks only reflect the extreme cases, we will only use the Criteo Terabyte for the subsequent studies.

C. Embedding Cache

We evaluate the effectiveness of the embedding cache by examining the hit rate, cache planning cost, and the number of accesses to SSD. The result is shown in Fig. 11. We separately show the hit rate in the main and hot-entry tables. For different batch sizes, the hit rate in the hot-entry table remains constant since the hit rate for the hot-entry table is determined by the statistical feature of the dataset. Meanwhile, the hit rate in the main table increases with the batch size. Overall, we achieved over 90% cache hit rate on all settings and even over 95% on the 16K batch size. Since the software-managed cache is fully associated, we can fully utilize the capacity of the embedding cache. The search cost increases with the batch size as there are more embedding vectors to search. However, the overall

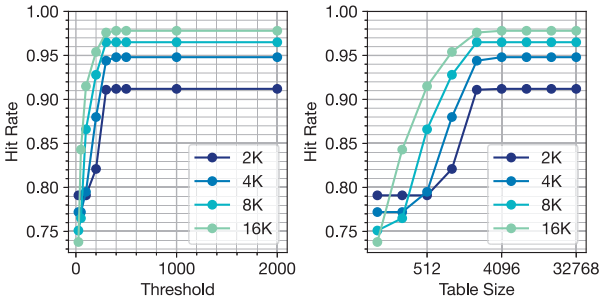

Fig. 12. The DRAM cache hit rate with different hot-entry thresholds (left) and hot-entry table sizes (right).

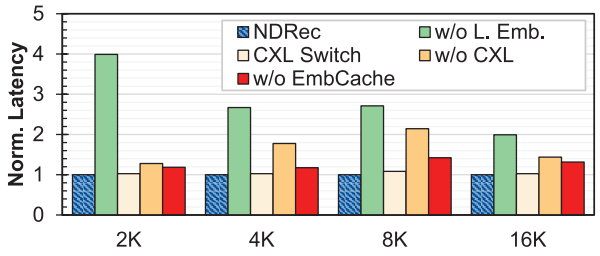

Fig. 13. The analysis of the contribution of each component.

search cost can still be overlapped by the computation latency. Regarding the SSD access, the embedding cache significantly reduces the read and write from/to SSD. The accesses to SSD are a mix of approximately 50% read and 50% write for EV loading and eviction. Even with the 16K batch size, there is only 9.18MB traffic between SSD and embedding cache with less than 12ms transfer latency per device. Since only the evicted entries are updated to SSD, the embedding cache significantly reduces the write to SSD, avoiding wearing out the SSD.

We further investigate the design choices for the hot-entry table, specifically, the selection of hot-entry threshold and hot-entry table size. The result is shown in Fig. 12. A small threshold could promote a less frequently accessed EV to the hot-entry table and prevent it from being evicted. A small hot-entry table size might lose the opportunity to capture the frequently accessed EVs. We also notice that, given the large capacity of the embedding cache and fully associated organization, the hit rate reaches a plateau when both parameters pass a certain value. Thus, we select 200 as the hot-entry threshold and 2048 as the hot-entry table size.

D. Ablation Study

Component Contribution. We perform an ablation study to evaluate the contribution of individual components in NDRec. The result is depicted in Fig. 13. We remove lookahead embedding, CXL, and embedding cache separately in our evaluated system. For the system without lookahead embedding, the training tasks are scheduled as shown in Fig. 4 'Naiv'. We see up to 4× slowdown since the fully sequential training timeline significantly increases the bandwidth requirement of the embedding operation. The gap narrows to 2× when we increase the batch

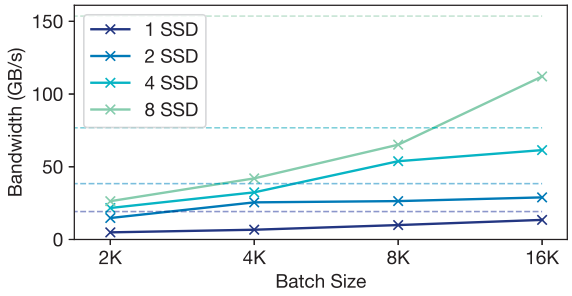

Fig. 14. Achieved aggregated bandwidth with different configurations. The dashed lines illustrate the theoretical maximum bandwidth.

size to 16K, as the cache design alleviates the bandwidth issue. For the system without CXL, we use DMA to communicate between GPU and CPU and refer to [8] to verify the achieved bandwidth in the simulator. To efficiently use DMA, we only initiate the data transfer when the embedding forward process is finished, which forces an additional synchronization barrier. The lack of CXL can cause up to 2.5× increment in latency due to the inefficient communication for the small amount of data. We also evaluate the case where two-level switching is required for SmartSSD to reach the host. Negligible overhead can be observed. For the system without embedding cache, we employ a simple static caching strategy where 3% of embedding parameters are cached in FPGA DRAM and randomly evict entries to load the necessary embedding vector for the new iteration. It increases the latency by 20% for 2K and 4K batch sizes, while up to 41% increment in latency can be observed for larger batch sizes. The simple static cache cannot effectively exploit the reuse opportunities.

E. Discussion

Bandwidth. We present the achieved aggregated bandwidth between the embedding cache and FPGA on SmartSSDs under different configurations in Fig. 14. The theoretical maximum bandwidth of the memory on a single SmartSSD device is 19.2GB/s. The achieved bandwidth is affected by multiple factors, including the efficiency of the memory controller, the size of EV loading required for each iteration, and the number of reused EVs in the on-chip buffer. A large batch size requires loading more EVs from the embedding cache, thus making it easier for the reader to schedule the memory request. The peak bandwidth approaches the efficiency of burst read as listed in [15]. For a small batch size, fewer EVs are required for each iteration. The access pattern is more random and reduces the efficiency of the memory controller. We notice that, for the 8K batch size, the 4 SSDs configuration achieves a similar bandwidth to that of the 8 SSDs configuration, indicating the bandwidth is not a limiting factor here. We can also tell that, for the configuration with less than 4 SmartSSDs, the achieved bandwidth with 16K batch size is limited by the device bandwidth. For further scaling up of the training batch size, the limited memory bandwidth could be the bottleneck for the performance. However, the configurations mentioned in [29] use the 16K batch size at most (2048 local batch size with 8

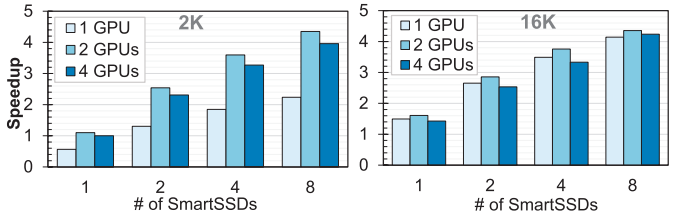

Fig. 15. Average speedup over GPU-UVM with 2K and 16K batch size and various combinations of GPUs and SmartSSDs.

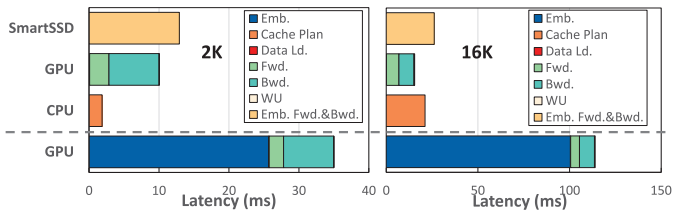

Fig. 16. The execution time breakdown of NDRec and GPU-UVM.

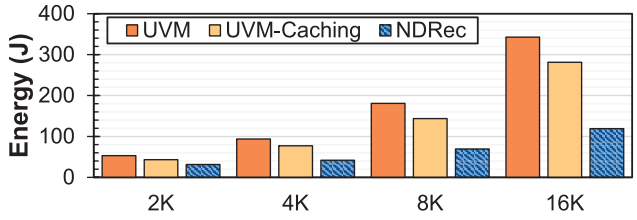

Fig. 17. The average energy consumption per iteration.

GPUs per node). Eight or more SmartSSDs should be able to serve one training node.

Scalability. We evaluate the scalability of the proposed NDRec system by scaling the number of GPUs and SmartSSDs. For GPU, we use weak scaling, i.e., the overall batch size remains constant so that we can maintain a similar amount of workload on SmartSSDs. Results are depicted in Fig. 15. When we have more SmartSSDs, the speedup ratio scales accordingly, showing the extra bandwidth/computing power provided by additional SmartSSDs can be efficiently utilized. We also noticed that when the number of SmartSSDs is smaller than or equal to 4, the speedup ratio scales linearly, indicating the overall performance is limited by the bandwidth or computing power of the SmartSSDs. Combined with the bandwidth analysis shown in Fig. 14, we can conclude that 4 SmartSSDs are sufficient to support the 8K batch size while 8 SmartSSDs are necessary for the 16K batch size. As for the GPUs, we can tell that for the 2K batch size, the GPUs form the bottleneck. Thus, increasing the number of GPUs from 1 to 2 shows nearly 2× improvement. For the 16K batch size, an additional GPU brings marginal benefit. In both cases, further increasing the number of GPUs to 4 narrows the gap over the baseline, indicating the baseline benefits from the extra GPUs while NDRec does not since the computation on GPU is no longer the bottleneck of the training.

Latency Breakdown. We present a latency breakdown of NDRec and GPU-UVM in Fig. 16. The result illustrates that the embedding operation dominates the training latency due to frequent data movement between the GPU and main memory. NDRec addresses this bottleneck by overlapping the embedding and other stages and avoiding expensive data movement.

Energy Consumption. We compare the energy consumption of NDRec per training iteration with the baselines. To estimate the energy consumption of the simulated part of NDRec, we use the measured worst-case dynamic power of the embedding kernel running on SmartSSD to obtain a pessimistic number for comparison. The result is illustrated in Fig. 17. NDRec can utilize the customized FPGA kernel and reduce the burden of data movement and GPU. On average, NDRec reduces the energy consumption by 54.9% and 43.8% over the UVM and UVM-Caching systems, respectively.

VII. Related Works

Near-Data Processing. NDP-based DLRM accelerators [19], [20], [23], [39], [40] build customized computing units near where the embedding tables are stored. TensorDIMM [23] and RecNMP [19], [20] directly process the embedding operation when the embedding vectors are read from the main memory. Customized instructions are used to trigger the computation, and the data layout is optimized to maximize the effective bandwidth. RecSSD [40] and RM-SSD [35] select storage as the level of operation. By modifying the FTL and/or architecture of SSD controllers, these designs are able to utilize the statistical information of embedding table access to optimize the data movement and fully utilize the internal bandwidth of the SSDs. Unlike NDRec, these solutions only support inference tasks. Even if they may be inserted into the training pipeline, the embedding operation is sequential with other processes and remains the bottleneck in the critical path. NDRec addresses these issues with relaxed dependencies through speculative embedding.

Multi-Tier Caching. The embedding table may be stored in main memory and/or storage and only necessary embedding vectors are retrieved during each iteration. cDLRM [7], ScratchPipe [24], and BagPipe [3] use a two-level storage (main memory and GPU memory). The data is prefetched to GPU memory with a caching strategy. AIBox [45] and RecShard [32] also include storage in the tiers. Apart from the caching strategy, the statistical features of the embedding vectors are used to determine the placement of data. These solution only offload the *data* rather than the *computation* of the embedding layer. The embedding layer remains on the critical path while the further inflation of DLRM model could saturate the available bandwidth of the system. NDRec performs computation at the storage level and only transfers the embedding output to GPUs, leading to higher utilization of GPU cores and alleviating the pressure on the system bus.

VIII. Conclusion

In this paper, we present NDRec, a near-data processing system for large-scale recommendation model training. We offload the embedding operation to SmartSSDs to enable the training

of large recommendation models and increase the utilization of GPUs. We then propose the lookahead embedding scheme to enable concurrent execution between GPU and CSDs. A software-managed DRAM cache and a customized FPGA kernel are designed to maximize the performance of the embedding operation. The evaluation result shows that NDRec could achieve up to 4.33× and 3.97× speedup over heterogeneous CPU-GPU platform and GPU caching, respectively. NDRec eliminates the capacity bottleneck and could enable the adoption of larger-scale recommendation models.

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 *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 802–814.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 33–42.

[3] S. Agarwal, Z. Zhang, and S. Venkataraman, "BagPipe: Accelerating deep recommendation model training," 2022, *arXiv:2202.12429*.

[4] T. Allen and R. Ge, "Characterizing power and performance of GPU memory access," in *Proc. 4th Int. Workshop Energy Efficient Supercomput. (E2SC)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 46–53.

[5] "Amazon personalize." Amazon Web Services. Accessed: Nov. 18, 2021. [Online]. Available: https://aws.amazon.com/personalize/

[6] R. Baeza-Yates, C. Hurtado, and M. Mendoza, "Query recommendation using query logs in search engines," in *Proc. Int. Conf. Extending Database Technol.*, Berlin, Germany: Springer-Verlag, 2004, pp. 588–596.

[7] K. Balasubramanian, A. Alshabanah, J. D. Choe, and M. Annavaram, "cDLRM: Look ahead caching for scalable training of recommendation models," in *Proc. 15th ACM Conf. Recommender Syst.*, 2021, pp. 263–272.

[8] R. Bittner and E. Ruf, "Direct GPU/FPGA communication via PCI express," in *Proc. 41st Int. Conf. Parallel Process. Workshops*, 2012, pp. 135–139.

[9] "UltraScale architecture memory resources user guide." Xilinx. Accessed: Jul. 27, 2022. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[10] "cDLRM." GitHub. Accessed: Oct. 5, 2022. [Online]. Available: https://github.com/lkp411/cDLRM

[11] H.-T. Cheng et al., "Wide and deep learning for recommender systems," in *Proc. 1st Workshop Deep Learn. Recommender Syst.*, 2016, pp. 7–10.

[12] "CMM-H (CXL Memory Module, H: Hybrid)." Samsung. Accessed: Jan. 5, 2024. [Online]. Available: https://samsungmsl.com/cmmh/

[13] "Compute Express Link (CXL) specification," Computer Express Link Consortium Inc., Beaverton, OR, USA, revision 3.0, Version 1.0, Aug. 2022. [Online]. Available: https://computeexpresslink.org/cxl-specification/

[14] "Download Criteo 1TB Click Logs dataset." Criteo AI Lab. Accessed: Nov. 18, 2021. [Online]. Available: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[15] "PG150—UltraScale architecture FPGAs memory IP product guide." AMD. Accessed: Jul. 27, 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf

[16] U. Gupta et al., "The architectural implications of Facebook's DNN-based personalized recommendation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 488–501.

[17] M. Jung, "Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD)," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, 2022, pp. 45–51.

[18] "Display advertising challenge." Kaggle. [Online]. Available: https://www.kaggle.com/c/criteo-display-ad-challenge

[19] L. Ke et al., "RecNMP: Accelerating personalized recommendation with near-memory processing," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 790–803.

[20] L. Ke et al., "Near-memory processing in action: Accelerating personalized recommendation with AxDIMM," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, Jan./Feb. 2022.

[21] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug. 2009.

[22] R. Kumar, B. Verma, and S. S. Rastogi, "Social popularity based SVD++ recommender system," *Int. J. Comput. Appl.*, vol. 87, no. 14, pp. 33–37, 2014.

[23] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 740–753.

[24] Y. Kwon and M. Rhu, "Training personalized recommendation systems from (GPU) scratch: Look forward not backwards," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 860–873.

[25] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA accelerated near-storage data analytics on SSD," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 110–113, Jul./Dec. 2020.

[26] H. Li et al., "Pond: CXL-based memory pooling systems for cloud platforms," 2022, *arXiv:2203.00241*.

[27] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 106–109, Jul./Dec. 2020.

[28] J. Lowe-Power et al., "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.

[29] D. Mudigere et al., "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *Proc. 49th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA: ACM, 2022, pp. 993–1011, doi: 10.1145/3470496.3533727.

[30] M. Naumov et al., "Deep learning recommendation model for personalization and recommendation systems," 2019, *arXiv:1906.00091*.

[31] E. Oldridge et al., "Merlin: A GPU accelerated recommendation framework," 2020. [Online]. Available: https://irsworkshop.github.io/2020/publications/paper_21_Oldridge_Merlin.pdf

[32] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "RecShard: Statistical feature-based memory optimization for industry-scale neural recommendation," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 344–358.

[33] "SmartSSD computational storage drive—Installation and user guide." AMD. Accessed: Jul. 27, 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1382-smartssd-csd

[34] Y. Song, A. M. Elkahky, and X. He, "Multi-rate deep learning for temporal recommendation," in *Proc. 39th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2016, pp. 909–912.

[35] 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 *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 1056–1070.

[36] "pytorch/torchrec: Pytorch domain library for recommendation systems." GitHub. Accessed: Oct. 5, 2022. [Online]. Available: https://github.com/pytorch/torchrec

[37] S. Van Doren, "Abstract—HOTI 2019: Compute express link," in *Proc. IEEE Symp. High-Perform. Interconnects (HOTI)*, Piscataway, NJ, USA: IEEE Press, 2019, p. 18.

[38] F. E. Walter, S. Battiston, and F. Schweitzer, "A model of a trust-based recommendation system on a social network," *Auton. Agents Multi-Agent Syst.*, vol. 16, no. 1, pp. 57–74, 2008.

[39] Y. Wang et al., "REREC: In-ReRAM acceleration with access-aware mapping for personalized recommendation," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2021, pp. 1–9.

[40] M. Wilkening et al., "RecSSD: Near data processing for solid state drive based recommendation inference," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 717–729.

[41] "PCIe Peer-to-Peer (P2P) - Xilinx XRT." GitHub. Accessed: Nov. 18, 2021. [Online]. Available: https://xilinx.github.io/XRT/master/html/p2p.html

[42] S.-P. Yang et al., "Overcoming the memory wall with CXL-enabled SSDs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2023, pp. 601–617.

[43] K. Yu, A. Schwaighofer, V. Tresp, X. Xu, and H.-P. Kriegel, "Probabilistic memory-based collaborative filtering," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 1, pp. 56–69, Jan. 2004.

[44] W. Zhao et al., "Distributed hierarchical GPU parameter server for massive scale deep learning ads systems," in *Proc. Mach. Learn. Syst.*, 2020, vol. 2, pp. 412–428.

[45] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "AIBox: CTR prediction model training on a single node," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.*, 2019, pp. 319–328.

Shiyu Li (Graduate Student Member, IEEE) received the B.Eng. degree in automation from Tsinghua University, Beijing, China, in 2019. He is currently working toward the Ph.D. degree with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA, supervised by Prof. Yiran Chen. His research interests include computer architecture, algorithm-hardware codesign of deep learning systems, and near-data processing.

Yitu Wang received the B.Eng. degree in microelectronics from Fudan University, Shanghai, China, in 2020. He is currently working toward the Ph.D. degree with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA, supervised by Prof. Yiran Chen. His research interest includes the near storage/memory architecture design for data-intensive applications.

Edward Hanson (Graduate Student Member, IEEE) received the B.S. degree in computer engineering from the University of Maryland, Baltimore County (UMBC), in 2019, and the Ph.D. degree in computer engineering from Duke University, in 2023, under the guidance of Prof. Yiran Chen. He was awarded the Meyerhoff Premier Scholarship. His research interests include optimizing machine learning systems by codesigning algorithm, compile-time software, and architecture.

Andrew Chang (Member, IEEE) received the M.S. degree in computer and electrical engineering from Rutgers University. He is a Principal System Architect with Samsung Memory Solution Lab over 25 years of industry experience. He has worked on microprocessor design at Intel and HP, networking processors at Juniper, and flash storage controllers at Violin Memory and Western Digital/Sandisk. He leads a team researching full-stack memory acceleration solutions for hyperscale data centers. His interests include heterogeneous computing, power efficiency, scalable memory architectures, software–hardware codesign, and new memory/accelerator interfaces.

Yang Seok Ki (Member, IEEE) received the bachelor's and master's degrees in computer engineering and the Ph.D. degree in electrical and computer engineering from Seoul National University. He completed the Engineering Leadership Professional Program at UC Berkeley. He is the Vice President of the Memory Solutions Lab, Samsung Semiconductor. Since 2011, he has been leding the advanced development projects, including SmartSSD, Key-Value SSD, CXL Memory Expander, and more. He led the NVMe Key Value Standard, SNIA Key Value API, and SNIA Computational Storage initiatives. He is a member of the Open Computing Project Future Technology Initiative. Previously, he worked with Oracle's Server Technology Group. Earlier, he researched HPC, grid, and Cloud computing at the USC and the UC San Diego.

Hai (Helen) Li (Fellow, IEEE) received the Ph.D. degree from Purdue University, in 2004. She is currently the Clare Boothe Luce Professor and the Chair of the Electrical and Computer Engineering Department, Duke University. Her current research interests include neuromorphic circuits and systems for brain-inspired computing, machine learning acceleration and trustworthy AI, conventional and emerging memory design and architecture, and software and hardware codesign. She was a recipient of the NSF Career Award (2012), DARPA Young Faculty Award (2013), TUM-IAS Hans Fischer Fellowship from Germany (2017), and ELATE Fellowship (2020). She received nine best paper awards and additional nine best paper nominations from international conferences. She is a Distinguished Lecturer of the IEEE CAS Society (2018–2019) and a Distinguished Speaker of ACM (2017–2020).

Yiran Chen (Fellow, IEEE) received the Ph.D. degree from Purdue University, in 2005. He is currently the John Cocke Distinguished Professor in electrical and computer engineering with Duke University, and serving as the Director of the NSF AI Institute for Edge Computing Leveraging the Next-generation Networks (Athena) and the NSF Industry-University Cooperative Research Center (IUCRC) for Alternative Sustainable and Intelligent Computing (ASIC), and the Co-Director of Duke Center for Computational Evolutionary Intelligence (DCEI). He received 11 best paper awards, 1 best poster award, and 15 best paper nominations from international conferences and workshops. He received numerous awards for his technical contributions and professional services such as the IEEE CASS Charles A. Desoer Technical Achievement Award and the IEEE Computer Society Edward J. McCluskey Technical Achievement Award. He has been the Distinguished Lecturer of IEEE CEDA and CAS.