RAP: Resource-aware Automated GPU Sharing for Multi-GPU Recommendation Model Training and Input Preprocessing

Zheng Wang
zhengwang@ucsd.edu
University of California,
San Diego, USA

Yuke Wang
yuke_wang@cs.ucsb.edu
University of California,
Santa Barbara, USA

Jiaqi Deng
jqdeng@ucsb.edu
University of California,
Santa Barbara, USA

Da Zheng
dzzhen@amazon.com
Amazon
USA

Ang Li
ang.li@pnnl.gov
Pacific Northwest National
Laboratory
Richland, Washington, USA

Yufei Ding
yufeiding@ucsd.edu
University of California,
San Diego, USA

Abstract

Ensuring high-quality recommendations for newly onboarded users requires the continuous retraining of Deep Learning Recommendation Models (DLRMs) with freshly generated data. To serve the online DLRM retraining, existing solutions use hundreds of CPU computing nodes designated for input preprocessing, causing significant power consumption that surpasses even the power usage of GPU trainers.

To this end, we propose **RAP**, an end-to-end DLRM training framework that supports **R**esource-aware **A**utomated GPU sharing for DLRM input **P**reprocessing and Training. The core idea of RAP is to accurately capture the remaining GPU computing resources during DLRM training for input preprocessing, achieving superior training efficiency without requiring additional resources. Specifically, RAP utilizes a co-running cost model to efficiently assess the costs of various input preprocessing operations, and it implements a resource-aware horizontal fusion technique that adaptively merges smaller kernels according to GPU availability, circumventing any interference with DLRM training. In addition, RAP leverages a heuristic searching algorithm that jointly optimizes both the input preprocessing graph mapping and the co-running schedule to maximize the end-to-end DLRM training throughput. The comprehensive evaluation shows that RAP achieves 2.09× speedup on average over the sequential GPU-based DLRM input preprocessing baseline. In addition, the end-to-end training throughput of RAP is only 1.71% lower than the ideal case, which has no input preprocessing overhead.

CCS Concepts: • **Information systems → Recommender systems**; • **Computer systems organization → Neural networks**.

Keywords: Deep learning recommendation models, AI training systems, Input Preprocessing

ACM Reference Format:
Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. 2024. RAP: Resource-aware Automated GPU Sharing for Multi-GPU Recommendation Model Training and Input Preprocessing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640406

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640406

1 Introduction

Deep learning recommendation models (DLRMs) have emerged as the critical backbone for numerous recommendation tasks, including advertising [10, 20] and search result ranking [13, 41], making them primary resource consumers in industry data centers [6, 16]. Contrary to traditional deep learning models that are typically trained *"Offline"* using preprocessed training data [12, 37], industrial DLRMs embrace an *"Online"* training approach, leveraging freshly generated data to continually update the models [40, 47]. Every second, during the serving of a DLRM model, a substantial amount of user behavior data is generated, collected, and preprocessed, priming it for immediate use in online DLRM retraining [53].

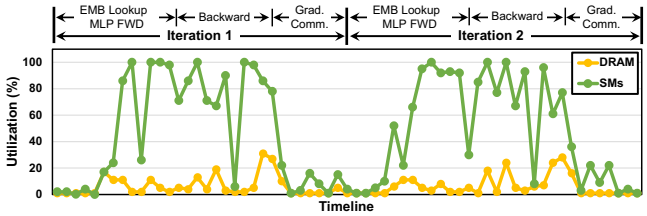
The online training paradigm empowers DLRMs with the capability to quickly adapt to dynamic shifts in user behavior distribution and the continuous generation of new content. However, this approach also introduces considerable challenges for designing DLRM training systems, particularly regarding input preprocessing. According to recent industry

workload analysis [53], a training job may require a substantial number of dedicated CPU computing nodes to meet its data consumption needs. Furthermore, it has shown that the input processing of DLRMs can lead to significant power and compute usage, even surpassing the power consumption of GPU trainers [53]. Surprisingly, recent DLRM optimization efforts [3, 5, 7, 26, 28, 48] concentrate on enhancing the training efficiency, seemingly assuming that online data preprocessing is readily available and does not require special attention.

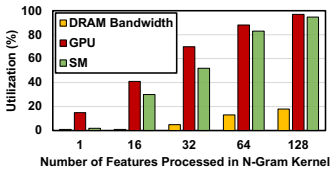
In contrast to the conventional approach of using dedicated nodes for preprocessing, our work presents an innovative alternative. We tap into the hidden potential of idle GPU resources during DLRMs training to perform input preprocessing while maintaining the original training throughput. Our key design insight is to leverage the unique characteristics of DLRMs, which integrate compute-intensive neural network layers with memory-intensive embedding layers, result in periodic variations in GPU resource utilization. Figure 1 (a) showcases the profiling results of DRAM bandwidth and SM utilization during two training iterations. Moreover, we go beyond resource optimization. By exploring the intricate supplier-consumer relationship between preprocessing and training workloads in DLRMs, we discover a promising opportunity for locality optimization to minimize unnecessary communication for inputs between GPUs. Our comprehensive approach not only maximizes the utilization of idle GPU resources during training but also fine-tunes the coordination between multi-GPU DLRM input preprocessing and training, resulting in enhanced overall training performance.

While placing preprocessing on GPUs shows promise, potential resource contention with training may hinder overall performance. To investigate this, we conduct a case study using the NGram input preprocessing operation[1]. Figure 1 (b) showcases that a larger kernel with more input features leads to higher GPU resource consumption. Additionally, Figure 1 (c) demonstrates that overlapping a large input preprocessing kernel (e.g., 128 input features) with DLRM training can cause large increases in training latency due to computing resource contention. Reflecting on this, we summarized three key challenges in co-allocating DLRM processing and training tasks. **Spatially,** mapping the preprocessing operations to multiple trainer GPUs presents a significant challenge. The optimal mapping strategy should consider both the data dependency to prevent unnecessary inter-GPU communication, and the disparities in the remaining GPU resources across GPUs, to minimize resource contention. **Temporally,** scheduling the co-running of input preprocessing with different DLRM training stages is challenging, due to the fluctuating GPU resource utilization during DLRM training and diverse GPU resource consumption patterns across different input

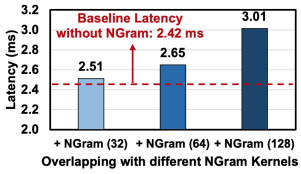
[1]An important feature generation operation for DLRM [53], which computes an n-gram across multiple sparse features to generate new input features



(a) A large amount of GPU resources are underutilized during DLRM training, which can be leveraged for input preprocessing.



(b) DRAM bandwidth utilization, GPU utilization, and SM utilization of the NGram input preprocessing kernel with different input sizes.



(c) Latency comparison when overlapping MLP forward with NGram kernel of different sizes. The latency increases when GPU resources are insufficient.

Figure 1. Opportunities and challenges of DLRM training and input preprocessing overlapping (each input feature of NGram kernel in (b) and (c) contains 4096 input samples).

preprocessing operations. **Jointly,** the interaction between the spatial input preprocessing operation mapping and the temporal co-running scheduling necessitates a holistic solution that co-optimizes both aspects for optimal performance, resulting in a significant search space.

To address the above challenges, we propose **RAP**, an end-to-end DLRM training framework that supports **R**esource-aware **A**utomated GPU sharing for DLRM input **P**reprocessing and Training. The key design insight of RAP is to accurately capture the remaining GPU computing resources during DLRM training and utilize them for DLRM input preprocessing in a fine-grained, resource-aware manner to achieve high end-to-end DLRM training efficiency without consuming additional computing resources. *Firstly,* to efficiently evaluate and select the appropriate spatial mapping plan, RAP offers a unified method for characterizing the cost of different input preprocessing operations and designs a co-running cost model (§5) to efficiently predict the performance of a given co-running plan. *Secondly,* to achieve optimal temporal co-running, RAP incorporates a resource-aware horizontal kernel fusion technique (§6). This technique adaptively fuses smaller preprocessing kernels based on remaining GPU resources, thereby avoiding interference with DLRM training. *Lastly,* RAP investigates the overlapping behaviors and the data-dependency of training and input preprocessing in-depth and designs a heuristic algorithm (§7) to jointly optimize the mapping of the input preprocessing graph and the co-running of DLRM training and input preprocessing.

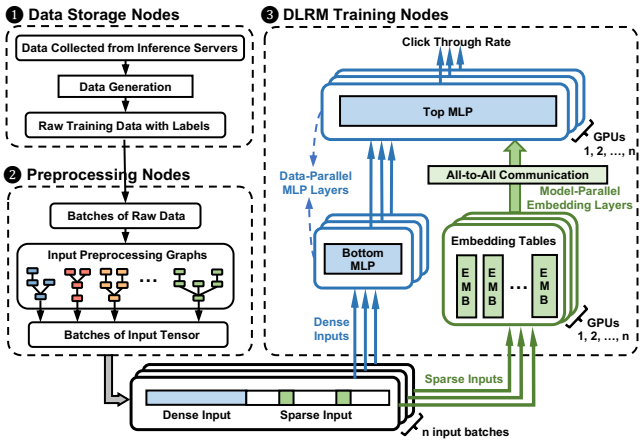


Figure 2. Overview of industrial DLRM training pipeline, including data storage nodes, input preprocessing nodes, and DLRM model training nodes.

Overall, we make the following contributions:

- To the best of our knowledge, we are the first to explore the potential of utilizing the leftover GPU resources from DLRM model training for input preprocessing.
- We propose RAP, an end-to-end DLRM training system that integrates both input preprocessing and DLRM model training, facilitating efficient online DLRMs training without consuming any additional computing resources beyond the trainer GPUs.
- Comprehensive experiments show that RAP achieves 2.09× speedup on average over the sequential GPU-based DLRM input preprocessing baseline and the end-to-end training throughput of RAP is only 1.71% lower than the ideal training throughput that has no input preprocessing.

2 Background

In this section, we will first provide the background of training and input preprocessing of DLRMs. Then we will introduce the basic of existing GPU multiplexing techniques.

2.1 Industrial DLRM Training Pipeline

In real-world applications of DLRMs, there is a continuous influx of new users and content, and the behaviors of existing users may shift over time [40, 47]. To maintain the quality of recommendations, it is crucial to continually update the model using new data generated from the inference servers [40]. To facilitate the online updating requirement of DLRM, a training pipeline – encompassing data storage, input preprocessing, and model training – is deployed in industrial DLRM applications [53, 54]. As shown in Figure 2, new data are collected from the inference servers, and stored in the *Data Storage Nodes*. Subsequently, the *Input Preprocessing Nodes* receive the raw data from the data storage nodes and apply a series of preprocessing operations to convert the raw data batch into input tensors. Lastly, these input tensors are sent to the *DLRM Training Nodes* for model retraining and updating. Distributing the data storage, input preprocessing, and model training to different computing nodes ensures the efficiency of each stage, but this results in higher cost and power consumption. Meta reported that the data storage and input preprocessing nodes account for over 50% of power consumption in its data centers, surpassing even the power usage of GPU trainers [53]. This motivates our design to utilize the remaining GPU resources on the training node for input preprocessing, instead of employing additional computing nodes.

Table 1. Common DLRM preprocessing operations. (DN/SN: Dense/Sparse Normalization, FG: Feature Generation)

Type	Operator	Description
DN	Logit	Logit transform for normalization
	BoxCox	BoxCox transform for normalization
	Onehot	Apply one hot encoding to normalize dense features
SN	SigridHash	Compute hash value to normalize list of sparse features
	FirstX	List truncation of sparse features for normalization
	Clamp	Clamp the sparse input based on the upper and lower bound
FG	Bucketize	Shard features based on bucket borders
	Ngram	Compute an n-gram between multiple sparse features
	Mapid	Maps feature IDs to fixed values
Others	FillNull	Fill NA/NaN values using the specified value
	Cast	Cast the data to different type

2.2 Hybrid Parallelism of DLRM

Different from traditional compute-intensive neural network architectures [19, 44], DLRMs incorporate not only the compute-intensive multi-layer perceptron (MLP) but also the memory-intensive embedding tables. The embedding tables in industrial DLRMs often contain billions of parameters, easily exceeding the memory capacity of a single GPU. To support the distinct architecture of DLRM, the state-of-the-art DLRM training systems adopt a hybrid parallelism training paradigm [3, 28, 39, 50, 51]. This method replicates the compute-intensive MLP layers on all GPUs for data parallelism. Concurrently, the large embedding tables are partitioned across GPUs and trained using model parallelism, as described in Figure 2 ❸. The hybrid parallelism training of DLRM leads to a unique data consumption flow. Specifically, each GPU requires the entire set of dense inputs within each input batch, as all GPUs have a duplication of MLP layers. Conversely, each GPU only processes a portion of the sparse input, since each GPU possesses only some of the embedding tables. This presents challenges when offloading input preprocessing to GPUs. If the input data needed by a GPU is not locally preprocessed, then additional input communication is required which inevitably decreases the throughput of training. This motivates the optimization for input preprocessing workload mapping to minimize input communication.

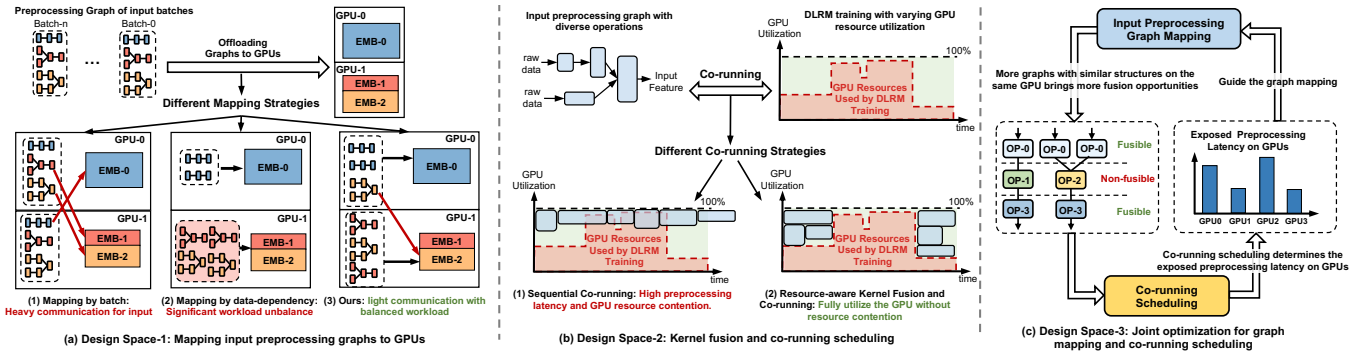


Figure 3. The scope and design space of RAP: (a) Design Space-1: The mapping of the input preprocessing graph, which influences inter-GPU communication and workload balance; (b) Design Space-2: The kernel fusion of input preprocessing and the co-running scheduling of DLRM training with input preprocessing to avoid resource contention. (c) Design Space-3: A joint optimization for input preprocessing mapping and co-running scheduling is required towards optimal end-to-end performance.

2.3 Input Preprocessing Operations of DLRM

As shown in Figure 2, DLRM has two types of input: *Dense Input*, which typically denotes continuous data (e.g., a user's age or login time) processed by MLPs, and *Sparse Input*, which represents discrete data (e.g., the category of an item) and is represented as one-hot or multi-hot binary vectors used to look up corresponding embeddings from the embedding tables. An input feature requires multiple preprocessing operations that can be represented as a directed acyclic graph (DAG) [53]. Table 1 lists the common input preprocessing operations of DLRMs. It can be observed that most input preprocessing operations for DLRM are element-wise and lightweight. This poses challenges for executing the input preprocessing operations on GPUs, as GPUs are designed for large, compute-intensive tasks like matrix multiplication [24] rather than small, diverse operations. Sequentially invoking small input preprocessing kernels on GPUs will result in significant kernel launching overhead and wastage of GPU computing resources. This motivates our resource-aware horizontal kernel fusion design to adaptively fuse small kernels for better resource utilization.

3 Motivation

Different from other traditional GPU sharing tasks [15, 49, 55] which aim to concurrently execute multiple small workloads on the same GPU to mitigate the GPU under-utilization issue, the task of overlapping DLRM training with its input data preprocessing has a unique objective and distinct patterns which cannot be adequately addressed by existing GPU sharing techniques. To highlight the unique challenges posed by the DLRM input preprocessing pipeline and to motivate the design of RAP, we illustrate the scope and design space of RAP in Figure 3. Specifically, optimizing the DLRM input preprocessing pipeline using the GPUs on the training nodes involves three major design spaces:

The mapping of the input preprocessing graph: As shown in Figure 3 (a), there are input preprocessing graphs of several input batches that need to be offloaded to the GPUs. The output of the input preprocessing graphs serves as the input data for the embedding tables (indicated by colors). Given this data-dependency correlation and the multi-GPU training paradigm inherent to DLRMs, the mapping of input preprocessing graphs could have a significant impact on end-to-end performance. We first discuss two straightforward heuristics for input preprocessing graph mapping. *(1) Mapping by batch:* This approach follows the input mapping of data parallel [38], in which the input preprocessing workload is divided batch-by-batch, with each GPU processing one batch of the input in a single iteration. However, this approach results in additional input data communication, as the input preprocessing graph and its corresponding embedding table are not co-located on the same GPU. *(2) Mapping by data-dependency:* Here, the input preprocessing workload is divided based on the placement of the embedding tables. Each GPU processes the corresponding input for its local embedding tables. Although *mapping by data-dependency* eliminates input communication, it leads to an imbalanced distribution of the input preprocessing workload. This result implies that the optimal input preprocessing graph mapping strategy should take into account both the input communication and the balancing of workloads, which motivates our data dependency-aware, workload-balance input preprocessing mapping strategy.

The co-running scheduling of DLRM training and input preprocessing: Following the mapping of the input preprocessing graph to the GPUs, the subsequent step involves scheduling the concurrent execution of input preprocessing and DLRM training. The co-running scheduling is challenging due to two reasons: *First*, the input preprocessing graphs involve diverse preprocessing operations which

have substantial differences in resource consumption and execution latency. For instance, the cost of feature generation operation is much higher than feature normalization [53]. *Second*, the remaining GPU resources from DLRM training vary significantly, which means we can not simply allocate a fixed amount of resources for input preprocessing. As shown in Figure 3 (b), the sequential co-running scheduling (e.g., Nvidia MPS [31]) will result in GPU resource contention and high preprocessing latency which inevitably influences the end-to-end DLRM training performance. To achieve efficient co-running of DLRM training and input preprocessing, we must fully explore kernel fusion opportunities to enhance the execution efficiency of input preprocessing kernels. At the same time, it is crucial to ensure that the resource consumption of the fused kernel does not exceed the available GPU resource limit.

Joint optimization for graph mapping and co-running scheduling: The mapping of preprocessing graphs and co-running scheduling are not independent of each other, necessitating a holistic approach to jointly optimize both aspects. As shown in Figure 3 (c), the mapping of input preprocessing graphs will influence the performance of co-running scheduling. If more graphs with similar structures are mapped onto the same GPU, it creates more opportunities for kernel fusion, thereby boosting input preprocessing efficiency. At the same time, the outcome of co-running scheduling determines whether there is exposed input preprocessing latency on GPUs. This crucial information can be harnessed to refine the mapping of the input preprocessing graph. The interaction between input preprocessing graph mapping and the co-running scheduling necessitates a joint optimization between them, rather than applying them independently.

4 Overview of RAP

Based on the above observation, we build RAP, which is an end-to-end DLRM training framework that efficiently eliminates the significant overhead of online data preprocessing in real-world DLRM training scenarios. This is achieved by offloading the input preprocessing computation to the DLRM training servers and pipelining the execution of training and input preprocessing on the same GPUs. To avoid data stalls and the interference between training and input preprocessing, RAP coordinates the co-running of DLRM training and input preprocessing in a resource-aware manner.

Figure 4 illustrates the workflow of RAP, which comprises two phases, offline and online. In the offline part, RAP gathers the execution latency data of all DLRM input preprocessing operations under varying configurations. This data then be used for training the *Preprocessing Latency Predictor* (Step ❶), which is subsequently utilized for online optimization.

In the online pass, RAP takes the configuration of a given DLRM training workload (including the computational graph

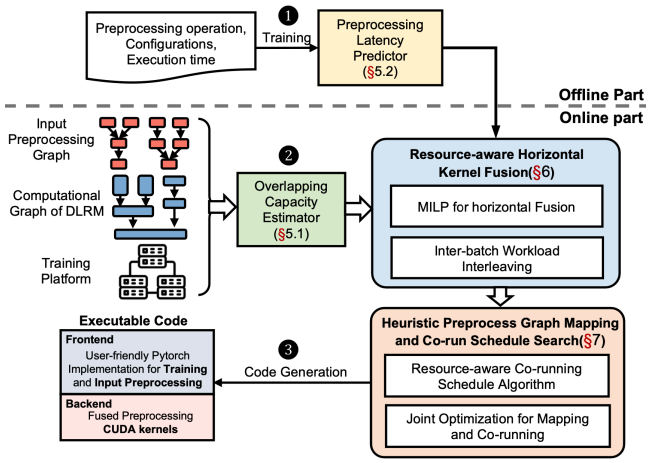


Figure 4. Overall workflow of RAP.

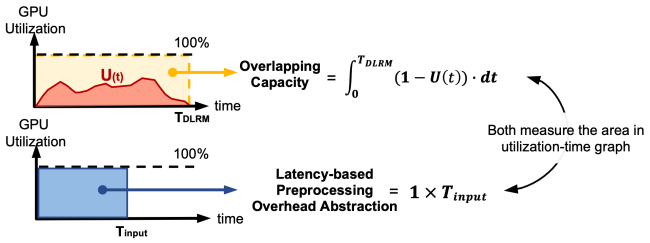
and the hardware information), along with the computational graph of data preprocessing as input. An *Overlapping Capacity Estimator* is then utilized to profile and evaluate the overlapping capacity of each DLRM training operation across various input preprocessing operations (Step ❷). Based on the estimated overlapping capacity, RAP then employs a heuristic algorithm to efficiently search for the optimal input preprocessing graph mapping and co-running scheduling for DLRM training and input preprocessing. Finally, RAP translates the searched plan into executable code, which includes optimized CUDA kernels and a user-friendly Pytorch frontend implementation (Step ❸).

5 Co-running Cost Model

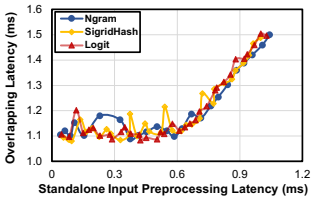
In this section, we will demonstrate our cost model design for the co-running of DLRM training and input preprocessing. Although actual hardware measurement can be used as a cost, it will be very time-consuming due to the significant search space introduced by the joint optimization of preprocessing graph mapping and co-run scheduling. Therefore, a lightweight cost model that efficiently predicts the performance of a given co-running plan is indispensable. Our cost model consists two components: *Overlapping Capacity Estimator* (§5.1) and *Preprocessing Latency Predictor* (§5.2).

5.1 Overlapping Capacity Estimator

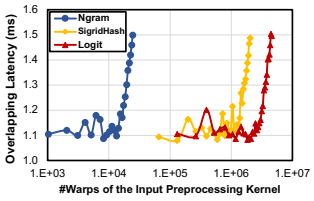
We first demonstrate our *overlapping capacity estimator* design. Assume the standalone DLRM training per-iteration latency is L_{Train}. When overlapping the input preprocessing pipeline with DLRM training, the end-to-end latency can be formulated as $\hat{L}_{Train} = L_{Train} + L_{\Delta}$. Here, L_{Δ} represents the increase in DLRM training latency caused by resource contention between DLRM training and input preprocessing. L_{Δ} can be eliminated by strategically coordinating the co-running of DLRM training and input preprocessing to



(a) Latency-based Preprocessing Overhead Abstraction: using the standalone execution latency of input preprocessing to bridge the overlapping capacity as they both measure the area in the utilization-time graph. ($U(t)$ is the relationship between GPU utilization and time, T_{DLRM} is the execution latency of DLRM training, and T_{input} is the execution latency of a given input preprocessing operation.)



(b) The correlation between standalone input preprocessing latency and overlapping latency. Different input preprocessing operations exhibit a similar trend.



(c) The cost of input preprocessing varies greatly: With the same #*warp*, different operations can lead to significant differences in overlapping latency.

Figure 5. The design insight and correctness verification of Latency-based Preprocessing Overhead Abstraction. (b) and (c) depict the correlation between overlapping latency and (1) standalone input preprocessing latency, and (2) the number of warps (#*warp*), respectively, for three typical preprocessing operations (Ngram, SigridHash, and Logit).

avoid resource contention. To achieve a contention-free co-running schedule, it is crucial to measure the overlapping capacity of different DLRM training operations (e.g., embedding lookup, MLP forward and backward). This metric measures the maximum number of input preprocessing operations that can be executed concurrently with the given DLRM training operation, without extending the total latency.

We propose a *Latency-based Preprocessing Overhead Abstraction* to connect the input preprocessing overhead with the overlapping capacity of DLRM training operations. As illustrated in Figure 5 (a), the overlapping capacity essentially represents the integration of the remaining GPU utilization over time. Similarly, during the standalone execution of the input preprocessing kernel, the GPU is fully occupied. Hence, both the overlapping capacity and the standalone input preprocessing latency measure the area in the utilization-time graph. To validate the effectiveness of *latency-based preprocessing overhead abstraction*, we measure the latency of embedding table lookup when overlapping with three types of preprocessing operations. As shown in Figure 5 (b), different input preprocessing operations exhibit a consistent trend when correlating standalone input preprocessing latency with overlapping latency. These findings suggest that standalone execution latency could serve as a uniform metric for modeling overlapping latency across a variety of input preprocessing operations. Figure 5 (c) further demonstrates the relationship between the number of warps (#*warp*) for input preprocessing operations and their overlapping latency. It can be observed that there is a noticeable misalignment between different curves, which indicates that the cost of each input preprocessing operation varies significantly. In conclusion, these results suggest that despite the significant variation in costs among different input preprocessing operations, the standalone execution latency of the input preprocessing kernel may reliably serve as a uniform metric to measure the overlapping latency.

5.2 ML-based Preprocessing Latency Predictor

Section 5.1 demonstrates that the standalone execution latency of input preprocessing operations can be harnessed to predict the end-to-end performance when co-running with DLRM training. This poses a new challenge: how can we efficiently determine the execution latency of input preprocessing kernels when searching for the optimal co-running plan of input preprocessing and DLRM training? Generally, the execution latency can be measured by testing the input preprocessing kernel on real hardware. However, this approach is not practical as it introduces significant profiling overhead at runtime. To overcome this challenge, we propose a *ML-based Preprocessing Latency Predictor* to efficiently predict the standalone execution latency of an arbitrary input preprocessing kernel. We choose XGBoost [9] as our predictor. It takes the type of the preprocessing operation and its corresponding configuration (e.g., input data size, output data size, and performance-related parameters) as input, and outputs the predicted execution latency. The process of training data collection and the predictor training are executed offline, which eliminates the runtime profiling overhead.

5.3 Co-running Cost Model Design

With our *overlapping capacity estimator* and *preprocessing latency predictor* design, the cost model of RAP is able to predict the performance cost of a given DLRM training and input preprocessing co-running plan. As depicted in Figure 6, the cost model takes a candidate co-running schedule as input. This schedule comprises two main parts: a DLRM training operation (MLP Forward in Figure 6) and several input preprocessing operations, which are assigned to overlap with the DLRM training operation. First, the *overlapping capacity estimator* profiles and yields the overlapping capacity, T_{OC}, of the given DLRM training operation. Given that the DLRM model remains unchanged across different co-running schedules, T_{OC} can be reused for other candidate schedules.

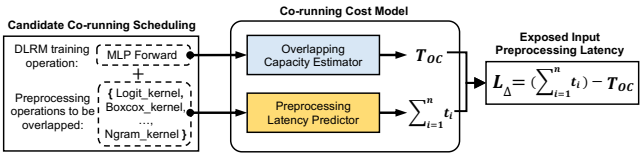


Figure 6. Workflow of Co-running Cost Model: the cost model takes the candidate co-running scheduling as input and outputs the exposed input preprocessing latency as the cost (T_{OC} is the overlapping capacity of the given DLRM training operation, t_i is the predicted execution latency of preprocessing operation i).

Consequently, the latency profiling for each DLRM training operation only needs to be conducted once. Next, the *preprocessing latency predictor* predicts the standalone execution latency of each input preprocessing operation, t_i, based on their configurations. It then sums up all execution latencies to obtain the total input preprocessing overhead, represented as $\sum_{i=1}^{n} t_i$. Finally, the cost model outputs the estimated cost of the given co-running schedule, which is calculated as the exposed input preprocessing latency: $L_\Delta = \sum_{i=1}^{n} t_i - T_{OC}$. From Figure 5 (b), it can be observed that, if $L_\Delta < 0$, the total execution latency will not increase. Therefore, our goal is to search for the optimal co-running schedule that minimizes L_Δ on all GPUs. Ideally, if the condition $L_\Delta < 0$ holds true for all GPUs, the performance of end-to-end DLRM training, encompassing both online input preprocessing and model training, would be equivalent to that of standalone DLRM model training.

6 Resource-aware Horizontal Kernel Fusion

In this section, we will first introduce the *horizontal kernel fusion* technique that fuses lightweight input preprocessing kernel to better utilize the remaining GPU resources from training. We then introduce a MILP formulation to efficiently search for the optimal horizontal fusion plan.

6.1 Horizontal Fusion for Preprocessing Kernels

DLRM input preprocessing operations are designed at the granularity of individual features, as different input features typically require unique preprocessing passes. However, this fine-grained execution of preprocessing operations will lead to GPU under-utilization as one single input preprocessing kernel is lightweight. To address this, we introduce a *Horizontal Fusion Technique* for input preprocessing kernels. Different from traditional kernel fusion techniques that combine kernels vertically to reduce the costly data round-trips to the GPU global memory [22, 46], our horizontal kernel fusion technique integrates multiple small kernels horizontally by allocating more threads to execute these kernels simultaneously. As illustrated in Figure 7, some input preprocessing operations are applied multiple times in the preprocessing

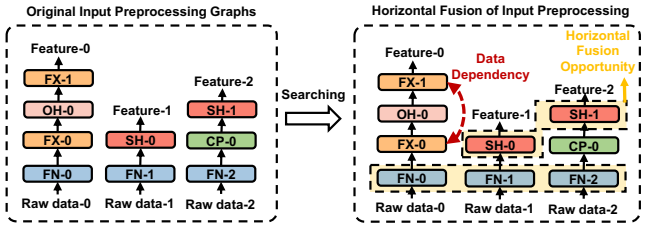


Figure 7. Illustration of the horizontal fusion opportunity. Different colors represent different input preprocessing operations (FN: FillNull, FX: FirstX, SH: SigridHash, CP: Clamp, OH: Onehot).

graphs, such as FillNull (FN), SigridHash (SH), and FirstX (FX). It can be observed that there is no data dependency between these FN and SH operations, allowing for their horizontal fusion. On the contrary, the two FX operations in this preprocessing graph exhibit a data dependency, with FX-1 requiring the completion of FX-0. Therefore, these two FirstX operations cannot be fused. Exploit all possible horizontal fusion opportunities within the input preprocessing graphs to maximize the performance gain is not trivial due to two reasons: (1) Horizontal kernel fusion is constrained by preprocessing type and data dependency. Only preprocessing operations of the same type with no data dependencies can be fused. (2) Horizontal fusion opportunities for different operations may conflict. For instance, if the preprocessing graphs contain both FirstX→SigridHash and SigridHash→FirstX sequences, a conflict arises when trying to apply horizontal fusion to both preprocessing operators.

6.2 MILP Formulation for Horizontal Fusion and Resource-aware Kernel Sharding

To overcome the challenges of searching for the optimal horizontal fusion plan, RAP formulates the problem as a mixed integer linear programming (MILP) [45] with quadratic terms in objective. By solving the problem using the MILP solver [17], RAP globally optimizes the horizontal fusion plan, while ensuring the data dependency constraint are not violated. Suppose there are N input preprocessing operations in total. We use a set $O = \{0, 1, \cdots, N-1\}$ to record the indices of all operations and a set $T = \{FillNull, Logit, \cdots, Ngram\}$ to record all possible preprocessing operation types. The set O can be divided into subsets according to the type of operations $O = O_{FillNull} \cup O_{Logit} \cup \cdots \cup O_{Ngram}$. We represent all potential horizontal fusion plans using a $N \times N$ binary matrix P. In this matrix, $P[i][j] = 1$ means that preprocessing operation-i is executed at time step-j. If multiple operations of the same type are assigned to the same time step, they will be horizontally fused into a single kernel. A special case arises when P is an identity matrix, which means no horizontal fusion since all operations are assigned to unique time steps. Based on this

representation, the constraints of MILP can be formulated as:

$$\sum_{j=0}^{N-1} P[i][j] = 1, \quad \forall i \in \{0, 1, \cdots, N-1\} \tag{1}$$

$$\sum_{k=0}^{N-1} (k+1) \cdot P[i][k] \geq \sum_{k=0}^{N-1} (k+1) \cdot P[j][k] + 1, \quad \forall i, j \in \{0, 1, \cdots, N-1\}, \text{and op-}i \text{ depends on op-}j \tag{2}$$

Equation 1 is a correctness constraint, ensuring that each input preprocessing operation is executed exactly once. And Equation 2 is a data-dependency constraint, guaranteeing that all input preprocessing operations are executed only after the operations they depend on have been completed.

The objective of the horizontal fusion problem is to maximize the fusion degree across all operation types within a given input preprocessing graph. It can be formulated as:

$$\textbf{Maximize} \sum_{Type} D_{Type}, \quad \forall Type \in T \tag{3}$$

$$D_{Type} = \sum_{j=0}^{N-1} (\sum_{i} P[i][j])^2, \quad \forall i \in O_{Type} \tag{4}$$

In Equation 4, $\sum_i P[i][j]$ quantifies the number of input preprocessing operations from O_{Type} are assigned to time step-j. In other words, it measures the degree of horizontal kernel fusion at time step-j. Our goal is to maximize the largest kernel fusion degree of all input preprocessing types. This objective can be achieved by maximizing the sum of the squares of all $\sum_i P[i][j]$ at different time steps, since $\sum_i P[i][j] \geq 0$ consistently holds.

The fused input preprocessing kernels searched by the MILP solver may be too large to co-run with a given DLRM training layer, since our MILP formulation for horizontal fusion aims solely to maximize the degree of horizontal fusion. To address this problem, we propose a *Resource-aware Fused Kernel Sharding* strategy that adaptively shards a fused kernel according to the available GPU resources. Specifically, before assigning a fused input preprocessing kernel to co-run with a particular DLRM training layer, RAP first leverages the *preprocessing latency predictor* (§5.2) to predict the standalone execution latency, T_{fuse}, of the fused kernel. If T_{fuse} is larger than the remaining overlapping capacity of the given DLRM training layer, RAP shards the kernel and reduces the kernel fusion degree until the kernel is small enough to co-run.

6.3 Inter-batch Workload Interleaving

The execution latency of DLRM input preprocessing is comprised of two parts: *GPU-side kernel execution latency* and *CPU-side data preparation latency*. Prior to the execution of the input preprocessing kernel, certain data preparation operations must be completed, including allocating memory space on the GPU for storing the result and transferring data from the CPU memory to the GPU memory for kernel execution. Rather than sequentially executing the data preparation and the preprocessing kernel, a more efficient solution is to co-run the preprocessing kernels with DLRM training layers that have more overlapping capacity and execute the data preparation concurrently with preprocessing kernels. However, the execution order of data preparation operations and the preprocessing kernels can not be changed arbitrarily, as there are data dependencies between them. To overcome this limitation, we propose an *Inter-batch Workload Interleaving* method. This approach separates the data preparation operations and the preprocessing kernels of different input batches and executes them in an interleaved manner. Specifically, we schedule the execution of preprocessing kernels for the current input batch and the data preparation for the next input batch within the same DLRM training iteration. Figure 8 shows the *inter-batch workload interleaving method*, which enables more flexible co-running scheduling to better utilize the GPU as it bypasses the data dependency between data preparation and the GPU kernel within the same data batch.

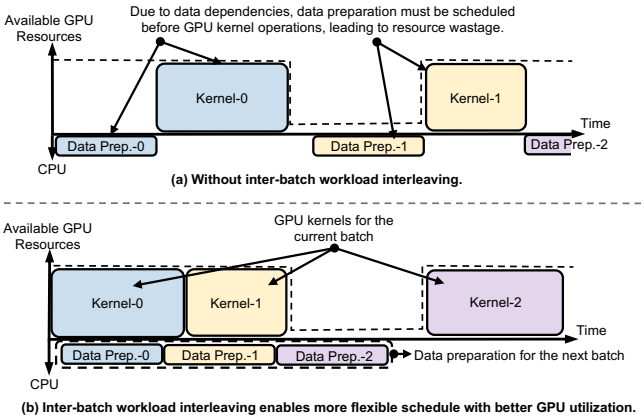


Figure 8. Illustration of inter-batch workload interleaving method which enables a more flexible co-running schedule to better utilize the GPU.

7 Heuristic Preprocess Graph Mapping and Co-run Schedule Search

In this section, we will demonstrate our heuristic algorithm design to jointly optimize both the inter-GPU level input preprocessing graph mapping and intra-GPU level co-running of DLRM training and input preprocessing.

7.1 Resource-aware Co-running Schedule Algorithm

Building on the horizontal fusion technique discussed in Section 6, which proactively adjusts the GPU resource consumption of input preprocessing operations, we propose a *Resource-aware Co-running Scheduling* approach. This method leverages the adaptability of horizontal fusion to optimize

the co-running of DLRM training and input preprocessing, targeting optimal end-to-end efficiency. It takes the input preprocessing graphs and the DLRM model as input and outputs the co-running schedule of DLRM training and input preprocessing. As detailed in Algorithm 1, RAP first employs the MILP solver [17] to search for the optimal horizontal fusion plan (Line 1). Secondly, RAP estimates the total input preprocessing latency of the fused kernels using the *preprocessing latency predictor* (Line 2-5). Based on the total input preprocessing latency, RAP selects the DLRM training layers based on their overlapping capacity, starting from the highest to the lowest, until the total overlapping capacity is sufficient for the input preprocessing kernels (Line 6-12). RAP then schedules the co-running in a greedy manner, following the order determined by the MILP solver (Line 13-29). Before assigning each kernel, RAP shards the kernel according to the available GPU resources to avoid potential resource contention between DLRM training and input preprocessing operations (Line 21-26). Following Algorithm 1, RAP can generate a co-running schedule for an arbitrary preprocessing graph assigned to each GPU, which avoids potential resource contention between input preprocessing and DLRM training.

Algorithm 1: Resource-aware Co-running Scheduling Algorithm.

```
   input  : Input Preprocessing Graphs: PG,
            DLRM Training Model: DLRM
   output : Co-running schedule: S
   /* Obtain optimal horizontal fusion using MILP solver.          */
1  Fused_Kernels = MILP_Solver(PG);
   /* Predict Input Preprocessing Latency.                         */
2  L = 0;
3  for kernel in Fused_Kernels do
4  |  L += Latency_Predictor(kernel);
5  end
   /* Sort the layer of DLRM by the overlapping capacity.          */
6  Sorted_DLRM = DLRM.sort_by_capacity()
   /* Select enough DLRM layers for kernel overlapping.            */
7  Layer_List = [];
8  for layer in Sorted_DLRM do
9  |  if Layer_List.total_capacity() < L then
10 |  |  Layer_List.append(layer);
11 |  end
12 end
   /* Schedule the co-running in a greedy manner.                  */
13 for layer in DLRM do
14 |  if layer in Layer_List then
15 |  |  while Fused_Kernels.size > 0 do
16 |  |  |  next_kernel = Fused_Kernels.pop();
17 |  |  |  Capacity = layer.capacity -
   |  |  |    S[layer].total_latency();
   |  |  |  /* If there is sufficient overlapping capacity.      */
18 |  |  |  if Capacity > Latency_Predictor(next_kernel) then
19 |  |  |  |  S[layer].append(next_kernel);
20 |  |  |  end
21 |  |  |  else
   |  |  |  |  /* Resource-ware kernel sharding.                 */
22 |  |  |  |  k_1, k_2 = next_kernel.shard(Capacity);
23 |  |  |  |  S[layer].append(k_1);
24 |  |  |  |  Fused_Kernel.push_front(k_2);
25 |  |  |  |  Break;
26 |  |  |  end
27 |  |  end
28 |  end
29 end
30 return S;
```

7.2 Joint Optimization for Preprocessing Graph Mapping and Co-run Schedule

Algorithm 1 addresses the intra-GPU level scheduling problem, offering an efficient method to search for the co-running schedule when input preprocessing graphs are mapped to a specific GPU. The remaining problem is determining the mapping of input preprocessing graphs across multiple GPUs. As described in Section 3, the mapping of input preprocessing graphs has a significant impact on the end-to-end performance as it influences both inter-GPU communication volume and workload balance. In addition, the mapping process at the inter-GPU level interacts intricately with the scheduling at the intra-GPU level, making the search for optimal input preprocessing graph mapping more challenging.

To address this problem, we propose a heuristic searching algorithm that jointly optimizes both the inter-GPU level input preprocessing graph mapping and the intra-GPU level co-running scheduling. The algorithm can be outlined in four steps: *First*, we use a data-locality-based approach for initial input preprocessing graph mapping. We map the input preprocessing graph based solely on the location of its data consumer. When a specific input feature is required by multiple GPUs (e.g., the input feature of the row-wise parallel embedding table), we duplicate the input preprocessing graph across all GPUs that require the data. This initial data-locality-based approach is optimal in input communication, as all input features are processed locally. *Second*, we evaluate the mapping based on the intra-GPU co-running schedule. Using Algorithm 1, we obtain the intra-GPU level co-running schedule based on the initial input preprocessing graph mapping. Then we evaluate the co-running schedule using our co-running cost model. *Third*, we generate a substitution mapping based on the estimated cost. This involves transferring a preprocessing graph from the GPU with the highest cost to the one with the lowest cost. Before making the transfer, we evaluate whether the move will enhance overall performance by weighing the benefits of improved workload balance against the potential increase in input communication costs. *Lastly*, we repeat the processes from the second and third steps until no further substitution mapping plan can be found.

Table 2. Detail of dataset and model architecture.

Dataset	Total Hash Size	Dimension	Dense Arch	Top Arch
Criteo Kaggle	33.7M	64	512-256	512-512-256
Criteo Terabyte	177.9M	128	512-256	1024-1024-512-256

Table 3. Detail of DLRM input preprocessing plan.

Plan	Dataset	#Dense Feature	#Sparse Feature	#Op per Feature	Total #Op
Plan 0	Kaggle	13	26	2.67	104
Plan 1	Terabyte	13	26	2.67	104
Plan 2	Terabyte	26	52	4.92	384
Plan 3	Terabyte	52	104	9.80	1548

8 Evaluation

In this section, we comprehensively evaluate *RAP* regarding the input preprocessing efficiency and the resulting benefits for online DLRM training.

8.1 Evaluation Setup

CPU-based Baseline: We compare *RAP* with ❶ *TorchArrow* [35, 36], which is a CPU-based Python DataFrame library designed for data preprocessing in deep learning. Notably, *TorchArrow* is currently the only data preprocessing framework that supports online input preprocessing for DLRMs. It has been employed in production-scale DLRM training of leading corporations such as Meta[35, 53]. In the evaluation, we use 8 input preprocessing workers per GPU to achieve higher input preprocessing throughput.

Handcrafted GPU-based Baseline: To demonstrate the advantages of *RAP*, we also implemented two GPU-based baselines utilizing two widely adopted GPU-sharing techniques, ❷ *CUDA stream* [32] and ❸ *Nvidia Multi-processing Service (MPS)* [31]. For *CUDA stream*, we initialize an additional stream with lower priority than DLRM training and assign the input preprocessing kernels to this stream. For *MPS*, we allocate two processes to each GPU: one for DLRM training and the other for input preprocessing. With *MPS* enabled, these two processes on the same GPU share the same CUDA context, enabling the overlapping execution of CUDA kernels.

Dataset: We choose two widely adopted DLRM datasets *Criteo Terabyte* [4] and *Criteo Kaggle* [1] for evaluation. *Criteo Terabyte* is the largest publicly available DLRM dataset. It has over four billion training samples that consist of feature values and click feedback of display ads. Each training sample contains 26 sparse features and 13 numerical features. *Criteo Kaggle* is the dataset for Criteo Kaggle Display Advertising Challenge which contains the records of Criteo's traffic spanning 7 days. The details of both the dataset and the corresponding model architecture are given in Table 2. The raw data is stored as column-based Apache Parquet files [34] in the disk and is loaded onto the GPU at runtime.

Input Preprocessing Plan: To comprehensively test the performance of *RAP* with different DLRM input preprocessing operations, we choose four input preprocessing plans. The detail of these three input preprocessing plans have been given in Table 3. For *Plan 0* and *Plan 1*, we follow the default input preprocessing plan for Criteo Terabyte dataset provided by TorchArrow. This default input preprocessing plan has a relatively low preprocessing density (number of operations per feature), consisting of FillNull operations applied to all input features, followed by certain normalization steps (e.g., SigridHash). To test *RAP* on larger datasets with more complicated input preprocessing graphs, we generate two additional preprocessing plans, *Plan 2* and *Plan 3*, by randomly applying different input preprocessing operations. These two plans have 2× and 4× more input features than *Plan 0* and *Plan 1*, respectively. Furthermore, these plans also have more preprocessing operations for each input feature.

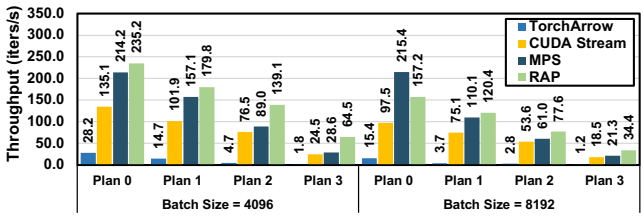
Platform & Tools: We leverage TorchRec (v0.3.2) [5], a PyTorch-based DLRM training framework, to implement the DLRM training component. The input preprocessing part of *RAP* is implemented using C++ and CUDA (v11.6). *RAP* automatically generates front-end Python code to call our optimized input preprocessing CUDA kernels and injects them into the TorchRec-based DLRM training program, making it user-friendly. To load raw data efficiently, we leverage CuDF (v21.8) which provides the API to directly load data from disk to GPU memory. Our major evaluation platform is Nvidia DGX-A100 [30], which incorporates 2× AMD EPYC 7742 64-Core Processor and 8× Nvidia A100 GPUs. Each GPU has 40GB memory and the GPUs are fully connected through NVSwitch and NVLink.

8.2 End-to-end Performance

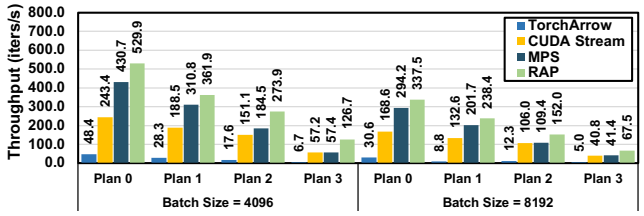
In end-to-end performance comparison, we run *RAP* and all baseline methods on diverse numbers of GPUs (from 2 to 8) with different input preprocessing plans and batch sizes.

Compared to *TorchArrow*: As shown in Figure 9, *RAP* consistently outperforms the CPU-based input preprocessing baseline *TorchArrow* significantly. On average, *RAP* achieves 18.4× speedup over *TorchArrow*. From the perspective of the number of GPUs, *TorchArrow* shows limited improvement when the number of GPUs increases. This is because the training pipeline is bottlenecked by the limited data preprocessing throughput on the CPU side. When CPUs have been fully utilized for input preprocessing, continuing to increase the number of GPUs will not bring improvement in the training throughput. In contrast, *RAP* demonstrates nearly linear improvement when scaling up to more GPUs. This is because the newly incorporated GPUs also have available resources for input preprocessing, which avoids data stalls when scaling to a larger number of GPUs.

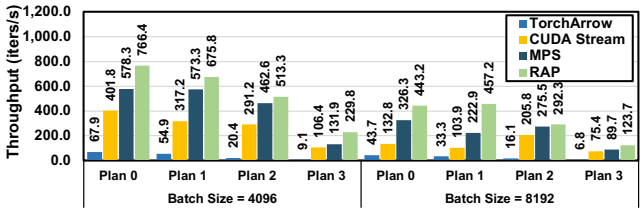
Compared to Handcrafted GPU-based Baselines: Utilizing remaining GPU resources from DLRM training for input preprocessing is not a trivial task. Handcrafted GPU-based baselines offload input preprocessing workloads to



(a) End-to-end DLRM training throughput on 2× A100 GPUs.



(b) End-to-end DLRM training throughput on 4× A100 GPUs.



(c) End-to-end DLRM training throughput on 8× A100 GPUs.

Figure 9. End-to-end DLRM training performance.

GPUs, employing two widely-adopted GPU sharing techniques, *CUDA stream* and *MPS*, to overlap input preprocessing with DLRM training. However, both approaches affect the throughput of DLRM training, leading to suboptimal end-to-end performance. Specifically, *RAP* achieves 2.01× and 1.39× speedup on average over the *CUDA stream* baseline and *MPS* baseline, respectively. The speedup mainly comes from our inter-GPU input preprocessing graph mapping and intra-GPU resource-aware overlapping scheduling optimization. Firstly, the handcrafted GPU-based baselines leverage the default data-parallel-based input preprocessing graph mapping which leads to input communications within the critical path of DLRM training. Secondly, the handcrafted GPU-based baselines schedule the execution of input preprocessing kernels sequentially, without considering the GPU resource constraints. This leads to resource contention between the DLRM training and the input preprocessing, which affects the DLRM training efficiency.

8.3 Optimization Analysis

Speedup Breakdown and Optimality Analysis: To better understand the benefits of individual optimizations and evaluate the gap from the optimal performance, we present a speedup breakdown and optimality analysis of *RAP* in Figure 10. Besides the full version of *RAP*, we create two additional settings of *RAP*: one without the inter-GPU input

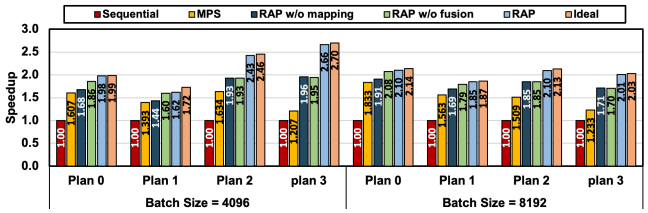


Figure 10. Speedup breakdown and optimality analysis.

processing graph mapping optimization (*RAP w/o mapping*), and another without the intra-GPU horizontal fusion optimization (*RAP w/o fusion*). We then compare these settings against three baselines: (1) *Sequential*, which executes DLRM training and input preprocessing sequentially, fully exposing all input preprocessing latency; (2) *MPS*, which leverages MPS to overlap input preprocessing with DLRM training; (3) *Ideal*, the ideal case with no input preprocessing and input communication. As shown in Figure 10, *RAP w/o mapping* and *RAP w/o fusion* deliver average speedups of 1.18× and 1.13× over *MPS*, respectively. This result demonstrates the effectiveness of each optimization. For the optimality analysis, it can be observed that the end-to-end performance of *RAP* is only 1.71% lower than the *Ideal* case. This implies that the input preprocessing are almost perfectly overlapped with DLRM training without compromising training efficiency.

Horizontal Fusion and Resource-aware Overlapping: To demonstrate the effectiveness of our horizontal fusion and resource-aware overlapping scheduling design, we fixed the DLRM training while gradually increasing the workload of input preprocessing by adding more Ngram preprocessing operations. And then we compare the end-to-end latency of three different settings: (1) *Baseline:* Offloading input preprocessing computation to GPUs without other optimization; (2) *Horizontal Fusion:* Enhancing the baseline by applying horizontal fusion; (3) *Fusion + Scheduling (RAP):* The full implementation of RAP, which integrates both horizontal fusion and resource-aware overlapping. In Figure 11, all three settings show a similar trend in DLRM training latency. The latency remains constant initially and then rises once the amount of preprocessing workload achieves a threshold. We denoted the turning point (where latency increases by more than 10%) on each curve using a vertical dashed line. It can be observed that the turning point of the *Baseline* arrives earliest. With horizontal fusion, multiple small preprocessing kernels fused together, reducing kernel launch overhead but enlarging individual kernels. This leads to higher GPU resource consumption and potential resource contention, causing the end-to-end latency to continue to increase even when the input preprocessing workload is not heavy. *RAP* employs a resource-aware overlapping scheduling method to address the GPU resources contention problem. By adaptively scheduling the co-run of DLRM training and input preprocessing based on their GPU resource consumption,

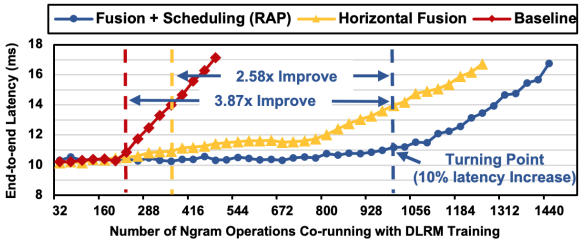


Figure 11. End-to-end DLRM training latency variation when input preprocessing workload increased.

Table 4. GPU and SM utilization at the turning point.

	Baseline	Horizontal Fusion	RAP
Avg. GPU Utilization	77.6%	79.3%	92.8%
Avg. SM Utilization	59.0%	66.7%	80.3%

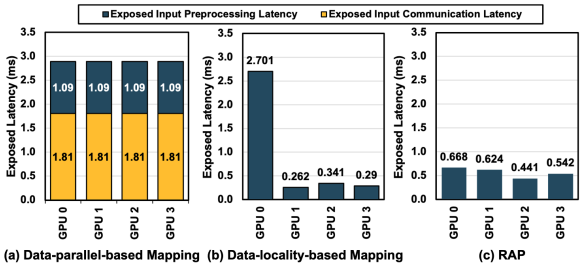


Figure 12. The exposed input communication latency and input preprocessing latency on GPUs with different input preprocessing graph mapping strategies.

Table 5. Accuracy of ML-based Processing Latency Predictor.

Operators	1D Ops	FirstX	Ngram	Onehot	Bucktize
Acc. (%)	98.0	95.5	92.9	97.3	98.5

RAP significantly delays the turning point of end-to-end latency.

GPU and SM Utilization: To demonstrate the advantage of *RAP* in better GPU resource utilization, we further profile the GPU and SM utilization of all three settings (*Baseline*, *Horizontal Fusion*, and *RAP* in Figure 11) at their respective latency turning points. The result has been given in Table 4. Both the GPU and SM utilization of the *Baseline* and *Horizontal Fusion* settings remained relatively low when the end-to-end DLRM training latency began to increase, indicating significant wastage of GPU resources under these settings. The resource-aware overlapping scheduling method employed by *RAP* helps to prevent potential GPU resource contention, resulting in significantly higher GPU and SM utilization.

8.4 Additional Study

Adaptability of Input Preprocessing Graph Mapping: We measure the exposed input preprocessing latency on GPUs when using three different input preprocessing graph mapping strategies on a skewed input preprocessing graph (embedding tables on GPU 0 has more input preprocessing operations): (1) *Data-parallel-based (DP) Mapping:* the input preprocessing workload is divided and mapped batch-by-batch; (2) *Data-locality-based (DL) Mapping:* the input preprocessing graphs are mapped according to the data dependency. Each GPU only processes the input feature it requires. (3) *RAP*: the input preprocessing graphs mapping strategy of *RAP*. As shown in Figure 12, the *DP mapping* exhibits the highest exposed latency, as it does not consider the data dependency of input, leading to input communication among GPUs. In contrast, *DL mapping* fully eliminates the input communication but leads to an imbalance in the distribution of input preprocessing workload. *RAP* takes a comprehensive approach to optimize input preprocessing graph mapping, considering both data locality and workload balance. It achieves 4.3× and 4.0× exposed latency reduction compared to *DP mapping* and *DL mapping*, respectively.

Accuracy of ML-based Preprocessing Latency Predictor To train and evaluate our machine learning-based preprocessing latency predictor, we gathered the execution latency of about 11K input preprocessing kernels with varying configurations. The data samples are randomly split into training and evaluation sets, following a ratio of 9:1. To achieve accurate latency predictions for various input preprocessing operators, we categorize these operators into five types. Specifically, for the *Ngram*, *Onehot*, *Bucketize*, and *FirstX* operations, which has unique performance-related parameters that can influence execution latency, we train a separate XGBoost model for each of them. In contrast, all other models are grouped as *1D Ops*, based on the observation that the preprocessing latency for these operators is primarily determined by the shape of the input data. We then evaluate the accuracy of our trained models by calculating the percentage of samples where the predicted latency deviates by no more than a 10% absolute gap from the actual measured latency. As listed in Table 5, all models demonstrate high prediction accuracy, with values ranging between 92.9% and 98.5%.

9 Related Work

DLRM training systems: Most existing DLRM training systems concentrate solely on the model training efficiency. For example, FAE [8] and RecShard [39] focus on minimizing the data movement between CPU and GPUs by caching the frequently accessed embeddings in GPU memory. TorchRec [5, 26] and HugeCTR [3] concentrate on optimizing the distributed training of large embedding tables using various embedding sharding techniques. Although these works achieve significant performance improvement in training throughput, they are all based on the assumption that the input data are already preprocessed offline, a scenario that does not

align with the online model updating paradigm in real-world DLRM applications [40]. In fact, the input preprocessing of industry-scale DLRMs is very costly, which may consume more power than training itself [53]. This calls for end-to-end DLRM training optimizations that incorporate both input preprocessing and DLRM training.

Data Preprocessing Pipeline for Deep Learning: The data preprocessing pipeline is a fundamental component in a variety of deep learning training frameworks [23, 27, 43]. As advancements in hardware and parallelization techniques continue to drive the rapid growth of deep learning model sizes and training efficiency [6], there is an increasing demand for higher input preprocessing throughput to meet the data consumption requirements of large-scale deep learning model training [21, 25, 43]. Recent efforts to address data stalls from inefficient input preprocessing pipelines fall into two broad categories. The first category offloads input preprocessing to remote nodes [14, 42, 43, 52, 53]. The downside of this approach is that it requires higher power consumption and costs as it utilizes additional computing nodes. Another direction is to offload the input preprocessing workload to specialized hardware, like FPGAs [11, 33]. However, this approach requires domain-specific expertise for manually optimizing input preprocessing operations on the specialized hardware. NVIDIA's Data Loading Library (DALI) [2] employs the GPUs for the input preprocessing of images and video workloads. However, since the input preprocessing and model training still operate sequentially, this inevitably leads to an increase in the end-to-end training latency. RAP concurrently executes DLRM training and input preprocessing on GPUs in a resource-aware manner, effectively avoiding any additional overhead for input preprocessing. Compared with previous preemption-based GPU sharing frameworks, like REEF [18], RAP generates an optimal overlapping schedule offline based on the data dependency relationships and resource requirements of input preprocessing and DLRM training, thereby eliminating the overhead associated with runtime preemption.

10 Discussion

Handling Runtime Variability of DLRM: The input distribution may shift over time [39], potentially affecting the performance of RAP. To address this issue, RAP could periodically generate a new fusion plan and overlapping schedule based on the current input distribution to maintain high training throughput. The regeneration process involves two steps: (1) Profiling the overlapping capacity of the embedding layers based on the new input distribution; (2) Searching for an optimal kernel fusion and overlapping plan adapted to the new input distribution. This regeneration process is lightweight, taking only a few minutes, which is negligible compared to the typical data shifting interval that spans days or months [39].

Extend RAP to Hybrid Input Preprocessing Approach: RAP aims to utilize the unused GPU resources remaining from DLRM training for input preprocessing. Based on our evaluation results, even the most costly input preprocessing workload (Plan 3) can be completely overlapped with DLRM training without increasing the per-iteration training latency. This result demonstrates that the remaining GPU resources from DLRM training are typically sufficient for input preprocessing in most cases. In scenarios where the input preprocessing workload is exceptionally intensive and the available GPU resources are limited, RAP can be adapted into a hybrid input preprocessing framework which employs both GPUs and CPUs for input preprocessing. This can be achieved by combining RAP with previous CPU-based input preprocessing frameworks like GoldMiner [52]. Specifically, RAP initially segments the input preprocessing graph into two distinct parts, designated for GPUs and CPUs respectively, taking into account the total capacity for overlapping on the GPUs. Subsequently, the portion of the graph allocated for CPUs can be processed by CPU-based input preprocessing frameworks. This strategic offloading will minimize CPU resource requirements while maintaining high end-to-end training efficiency.

11 Conclusion

This paper presents RAP, a novel end-to-end DLRM training framework that supports resource-aware automated GPU sharing for DLRM input preprocessing and model training. At the intra-GPU level, RAP proposes a horizontal kernel fusion technique to fully utilize the varying GPU resources leftover from DLRM training, and uses the MILP formulation to efficiently search for the optimal horizontal fusion plan. At the inter-GPU level, RAP employs a heuristic algorithm to jointly optimize the mapping of the input preprocessing graph and the co-running of DLRM training and input preprocessing. Comprehensive experiments demonstrate that RAP outperforms the state-of-the-art CPU-based DLRM input preprocessing framework and achieves near-perfect overlapping of input preprocessing and DLRM training.

12 Acknowledgment

We would like to express our appreciation for the great help and invaluable suggestions from the ASPLOS anonymous reviewers and shepherd. This work was supported in part by NSF 2124039 and CloudBank [29]. Additionally, this research was supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830. Also, we would like to thank the generous help and support from Amazon Faculty Research Award 2021 for Professor Yufei Ding.

References

[1] Criteo display ad challenge. https://www.kaggle.com/c/criteodisplay-ad-challenge.

[2] Nvidia data loading library (dali). https://developer.nvidia.com/dali.

[3] Nvidia merlin hugectr. https://developer.nvidia.com/nvidia-merlin/hugectr.

[4] Terabyte click logs. https://labs.criteo.com/2013/12/downloadterabyte-click-logs.

[5] Torchrec. github.com/pytorch/torchrec/, 2022.

[6] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021. https://doi.org/10.1109/HPCA51647.2021.00072.

[7] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, 2021. http://www.vldb.org/pvldb/vol15/p127-mahajan.pdf.

[8] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment*, 15(1):127–140, 2021. http://www.vldb.org/pvldb/vol15/p127-mahajan.pdf.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016. https://doi.org/10.1145/2939672.2939785.

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In Alexandros Karatzoglou, Balázs Hidasi, Domonkos Tikk, Oren Sar Shalom, Haggai Roitman, Bracha Shapira, and Lior Rokach, editors, *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, pages 7–10. ACM, 2016. https://doi.org/10.1145/2988450.2988454.

[11] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jiaxin Lin, Xi Fan, Jinkun Geng, Xinyi Yu, Wei Bai, Lei Qu, Ran Shu, Peng Cheng, Yongqiang Xiong, and Jianping 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, Kyoto, Japan, August 05-08, 2019*, pages 88:1–88:11. ACM, 2019. https://doi.org/10.1145/3337821.3337892.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. https://doi.org/10.18653/v1/n19-1423.

[13] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015. https://doi.org/10.1145/2843948.

[14] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022. https://www.usenix.org/conference/atc22/presentation/graur.

[15] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *NSDI*, volume 19, pages 485–500, 2019. https://www.usenix.org/conference/nsdi19/presentation/gu.

[16] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, pages 488–501. IEEE, 2020. https://doi.org/10.1109/HPCA47549.2020.00047.

[17] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. https://www.gurobi.com/documentation/current/refman/index.html.

[18] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated DNN inferences. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 539–558. USENIX Association, 2022. https://www.usenix.org/conference/osdi22/presentation/han.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. https://doi.org/10.1109/CVPR.2016.90.

[20] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017. https://doi.org/10.1145/3038912.3052569.

[21] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1825–1839, 2022. https://doi.org/10.1145/3514221.3517848.

[22] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019. https://doi.org/10.1145/3341301.3359630.

[23] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *Proc. VLDB Endow.*, 14(2):87–100, 2020. http://www.vldb.org/pvldb/vol14/p87-kang.pdf.

[24] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning gemm for gpus. In *Computational Science–ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I 9*, pages 884–892. Springer, 2009. https://doi.org/10.1007/978-3-642-01970-8_89.

[25] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.*, 14(5):771–784, 2021. http://www.vldb.org/pvldb/vol14/p771-mohan.pdf.

[26] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews,

Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 993–1011. ACM, 2022. https://doi.org/10.1145/3470496.3533727.

[27] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, 2021. http://www.vldb.org/pvldb/vol14/p2945-klimovic.pdf.

[28] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. http://arxiv.org/abs/1906.00091.

[29] Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, et al. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing*, pages 1–4. 2021.

[30] Nvidia. Nvidia dgx a100. www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[31] NVIDIA. Nvidia multi-process service. docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[32] Nvidia. Cuda c/c++ streams and concurrency. "http://on-demand.gputechconf.com/gtcexpress/2011/presentations/StreamsAndConcurrencyWebinar.pdf", 2011.

[33] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: An extreme-scale neural network training server architecture by systematically balancing operations. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 825–838. IEEE, 2020. https://doi.org/10.1109/MICRO50266.2020.00072.

[34] Apache Parquet. Apache parquet. https://parquet.apache.org/.

[35] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. Velox: Meta's unified execution engine. *Proc. VLDB Endow.*, 15(12):3372–3384, 2022. https://www.vldb.org/pvldb/vol15/p3372-pedreira.pdf.

[36] Pytorch. Torcharrow. https://pytorch.org/torcharrow/beta/index.html.

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[38] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020. https://doi.org/10.1109/SC41405.2020.00024.

[39] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 344–358. ACM, 2022. https://doi.org/10.1145/3503222.3507777.

[40] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 821–839. USENIX Association, 2022. https://www.usenix.org/conference/osdi22/presentation/sima.

[41] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017. https://doi.org/10.1109/MIC.2017.72.

[42] TensorFlow. Module: tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service.

[43] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proceedings of the VLDB Endowment*, 16(5):1086–1099, 2023. https://www.vldb.org/pvldb/vol16/p1086-um.pdf.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[45] Juan Pablo Vielma. Mixed integer linear programming formulation techniques. *Siam Review*, 57(1):3–57, 2015. https://doi.org/10.1137/130915303.

[46] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010. https://doi.org/10.1109/GreenCom-CPSCom.2010.102.

[47] Shoujin Wang, Longbing Cao, Yan Wang, Quan Z Sheng, Mehmet A Orgun, and Defu Lian. A survey on session-based recommender systems. *ACM Computing Surveys (CSUR)*, 54(7):1–38, 2021. https://doi.org/10.1145/3465401.

[48] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. El-rec: efficient large-scale recommendation model training via tensor-train embedding table. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1007–1020. IEEE Computer Society, 2022. https://doi.org/10.1109/SC41404.2022.00075.

[49] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2:98–111, 2020. https://proceedings.mlsys.org/book/294.pdf.

[50] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. Autoshard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4461–4471, 2022. https://doi.org/10.1145/3534678.3539034.

[51] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. Dreamshard: Generalizable embedding table placement for recommender systems. In *NeurIPS*, 2022. http://papers.nips.cc/paper_files/paper/2022/hash/62302a24b04589f9f9cdd5b02c344b6c-Abstract-Conference.html.

[52] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proc. ACM Manag. Data*, 1(2):193:1–193:25, 2023. https://doi.org/10.1145/3589773.

[53] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 1042–1057. ACM, 2022. https://doi.org/10.1145/3470496.3533044.

[54] Mark Zhao, Dhruv Choudhary, Devashish Tyagi, Ajay Somani, Max Kaplan, Sung-Han Lin, Sarunya Pumma, Jongsoo Park, Aarti Basant, Niket Agarwal, Carole-Jean Wu, and Christos Kozyrakis. Recd: Deduplication for end-to-end deep learning recommendation model training infrastructure. *CoRR*, abs/2211.05239, 2022. https://doi.org/10.48550/arXiv.2211.05239.

[55] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803*, 2023. https://doi.org/10.48550/arXiv.2303.13803.

A Artifact Appendix

A.1 Abstract

RAP represents an advanced end-to-end DLRM online training framework which enables resource-aware, automated GPU sharing for DLRM input preprocessing and model training. RAP automatically generates highly optimized code tailored for online DLRM training based on the given input preprocessing plan and the DLRM model configuration. The input preprocessing and DLRM model training will be seamlessly overlapped and concurrently executed on GPUs.

A.2 Artifact check-list (meta-information)

- **Hardware:** 8× A100 GPUs
- **Experiments:** The result of Figure-9 and Figure-10
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 1 hours
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access. The project is open-sourced at Github[2].

A.3.2 Hardware dependencies. To reproduce the results presented in the paper, we recommend to use a machine with 8× NVIDIA A100 GPUs (e.g. AWS p4d.24xlarge instance).

A.3.3 Software dependencies.

- TorchRec (v0.3.2)
- TorchArrow (v0.2.0a0)
- CuDF (v21.8)
- CUDA (v11.6)
- PyTorch (v1.13.1)
- Gurobi Solver

The required software dependencies have been included in our GitHub repository and the Docker image we provided.

A.4 Installation

1. To ease the setup process for experiments, we provide a Docker image.
2. Users can follow the instructions given in the README in our GitHub repository to pull and launch the Docker container and install the required dependencies.

A.5 Evaluation and expected results

We provide the scripts to reproduce the results presents in Figure-9 and Figure-10:

Figure-9: End-to-end DLRM training performance.

1. The codes and scripts needed to reproduce the results of the baseline (including TorchArrow, CUDA stream, and MPS) in Figure-9 are available in the '*RAP/baseline_end_to_end/*' directory.
2. By running the scripts, the results of the training throughput will be outputted to the '*result/*' directory.

Figure-10: Speedup breakdown and optimality analysis.

1. The codes and scripts needed to reproduce the results of the baseline (including Sequential, MPS, RAP w/o Mapping, RAP w/o Fusion) and ideal case in Figure-9 are available in the '*breakdown_study*' directory.
2. By running the scripts, the results of the training throughput will be outputted to the '*result/*' directory.

RAP: The implementation of RAP can be found in the directory '*/RAP/RAP_end_to_end*'. By running the script, RAP will automatically generate optimized code for DLRM training. The output code is located in '*combined_code/*'.

[2] https://github.com/Ash-Zheng/RAP-artifacts