Accelerating Recommendation System Training by Leveraging Popular Choices

by

Yassaman Ebrahimzadeh Maboud

B.A.Sc, Shahid Beheshti University (IRAN), 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled: "Accelerating Recommendation System Training by Leveraging Popular Choices" submitted by Yassaman Ebrahimzadeh Maboud in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical and Computer Engineering.

Examining Committee:

Prashant J.Nair, Assistant Professor, Electrical and Computer Engineering Department, UBC
Supervisor

Alexandra Fedorova, Professor, Electrical and Computer Engineering Department, UBC
Supervisory Committee Member

Satish Gopalakrishnan, Associate Professor, Electrical and Computer Engineering Department, UBC
Supervisory Committee Member

Abstract

Recommendation systems have been deployed in e-commerce and online advertising to expose desired items from the user's perspective. To meet this end, various deep learning-based recommendation models have been employed such as the Deep learning recommendation model or DLRM at Facebook.

The input of such a model can be categorized as dense and sparse representations. The former demonstrates the numerical representation of items and users with discrete parameters. On the other hand, the latter refers to continuous input such as time or age. Such models are comprised of two main components: computation-intensive components like multilayer perceptron or MLP and memory-intensive like embedding tables which save the numerical representation of sparse. Training these large-scale recommendation models is evolving to require increasing data and compute resources.

The highly parallel neural networks portion of these models can benefit from GPU acceleration, however, large embedding tables often cannot fit in the limited-capacity GPU device memory. Hence, this thesis deep dives into the semantics of training data and feature access, transfer, and usage patterns of these models. We observe that, due to the popularity of certain inputs, the accesses to the embeddings are highly skewed. Only a few embedding entries are accessed up to 10000× more.

In this thesis, we focus on improving the end-to-end training performance using this insight and offer a framework, called Frequently Accessed Embeddings or FAE. we propose a hot-embedding-aware data layout for training recommender models. This layout utilizes the scarce GPU memory for storing the highly accessed embeddings, thus reducing the data transfers from CPU to GPU. We choose DLRM [41] and XDL [30] as the baseline. Both of these models have been com-

mercialized and are well-established in the industry. DLRM has been deployed by Facebook as well as XDL by Alibaba. We choose XDL as of its high utilization of CPU and a notably scalable solution for training recommendation models.

Experiments on production-scale recommendation models with datasets from real work show that FAE reduces the overall training time by 2.3× and 1.52× in comparison to XDL CPU-only and XDL CPU-GPU execution while maintaining baseline accuracy.

Lay Summary

The recent trend for recommendation model shows that training these model demands higher computation power and memory capacity. One approach to tackle this is distributed training by deploying CPU along with GPU. The baseline, deployed in corporations such as Facebook, handles the issue by exploiting data parallelism and model parallelism. This approach incurs a significant overhead of CPU-GPU communication for each iteration. To overcome the issue, a more enhanced way has been suggested by XDL. It has been used as our second baseline.

In this thesis, we propose a framework, Frequently Accessed Embeddings or FAE to mitigate the GPU-CPU communication overhead by leveraging the fact that entries in the embedding tables are not accessed evenly. We demonstrate our results acquiring 4.76 × and 1.80× against the open-source DLRM and TBSM on CPU and CPU-GPU, and 2.3× and 1.52× in comparison to XDL CPU-only and XDL CPU-GPU execution while maintaining baseline accuracy.

Preface

This thesis is the result of work carried out by me, under supervision of my advisor, Dr. Prashant Nair, and Dr. Divya Mahajan (Microsoft Research). All chapters are based on paper published in the 2022 International Conference on Very Large Data Bases (VLDB). I was responsible for conceiving the ideas. My colleague Muhammad Adnan and I were responsible for designing and conducting the experiments and writing the paper. Dr. Nair was responsible for overseeing the project, providing guidance and feedback, and editing and writing parts of the paper.

Dr. Mahajan contributed her knowledge and expertise due to her closely related prior work. She provided insights into the problems, helped in designing experiments, and assisted with the tools used for these experiments.

Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, Prashant Nair. "Accelerating Recommendation System Trainingby Leveraging Popular Choices". In *International Conference on Very Large Data Bases (VLDB), 2022*.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit
DLRM	Deep Learning Recommendation Model
FAE	Frequently Accessed Embeddings
GPU	Graphics Processing Unit
MLP	Multilayer Perceptron

Acknowledgments

I would like to express my sincere gratitude to my advisor Dr. Prashant Nair for his perpetual support during my Masters. His patience, invaluable guidance, and in-depth knowledge made my stay UBC extremely rewarding and helped me grow personally as well as professionally. His enthusiasm and an optimistic outlook and patience are quite inspiring for young researchers like me.

I would like to mention and Thank my mentor Dr. Divya Mahajan for her insightful comments and never-ending support throughout the whole process.

I would like to thank my colleague Muhammad Adnan and many other colleagues in the Gattaca Lab at UBC for their support and stimulating discussions throughout my research endeavors.

Finally, I would like to thank my family, for providing me with constant encouragement and having my back throughout my study. This degree would not have been possible without the zealousness and sincere support from them.

Chapter 1

Introduction

1.1 Recommendation Model overview

Recommendation models are an important class of machine learning algorithms that enable the industry (Netflix [18], Facebook [41],Amazon [53], etc.) to offer a targeted user experience through personalized recommendations. Deep learning based recommendation models [26, 41] are at the core of a wide variety of internet services and consume significant infrastructure capacity and compute cycles in data centers [42]. Training such at-scale models observes a conflation of challenges arising from high compute and data storage/transfer requirements.

On the compute side, hardware accelerators notably GPUs and other heterogeneous architectures [7, 8, 10, 31, 37], provide a robust mechanism to increase performance and energy efficiency. To mitigate the large memory requirement, distributing training load through model parallel training or reducing the overall memory requirement through sparsity and compression can be used. However, such techniques either require a pool of hardware accelerators that cumulatively provide enough memory to store these large models or trade off accuracy from the reduced precision for model footprint.

1.2 Motivation

Recommender models, as shown in Figure 1.1, use embedding tables that contribute heavily towards the memory capacity requirement and neural networks that exhibit compute intensity. While neural networks can benefit from GPUs, embedding tables (10s of GBs) often cannot fit within GPU memories [24, 42]. Naively using model parallelism just to store the large embedding data across multiple GPUs is sub-optimal, as the number of GPU devices per compute node are not only fixed, but also scarce and expensive. Figure 1.1 shows the size of the embedding tables for four real-world datasets across two open-source recommender models, "Deep Learning Recommendation Model for Personalization and Recommendation Systems" (DLRM) and "Time-based Sequence Model for Personalization and Recommendation Systems"(TBSM).

As user-targeted applications evolve, the size of these embedding tables is expected to increase at a rate faster than the anticipated increase in the memory capacity. This is because larger embedding tables can track a greater and more diverse degree of user preferences. Therefore, in practice, it is common to train recommendation models either solely on CPUs or use the CPUs for handling the embedding data with GPUs executing data-parallel neural networks. In the latter case, embeddings are stored in CPU memories as shown in Figure 1.1 and require embedding data to be transferred between CPU and GPUs.

Past work has shown that data transfers are not only performance degrading but also consume significantly higher energy compared to accessing memories on the device. To address this, we leverage the observation that certain embedding entries and inputs to recommendation models are significantly more popular than the others. For instance, blockbuster movies tends to be significantly more popular than other movies. Each of these items or users are translated to an entry to the embedding table. Therefore, when an item is popular, the associated entry is used more in the model as well. Below, we discuss how popular inputs and embeddings can be delegated to a faster and compute-proximate device memory while maintaining the training fidelity.

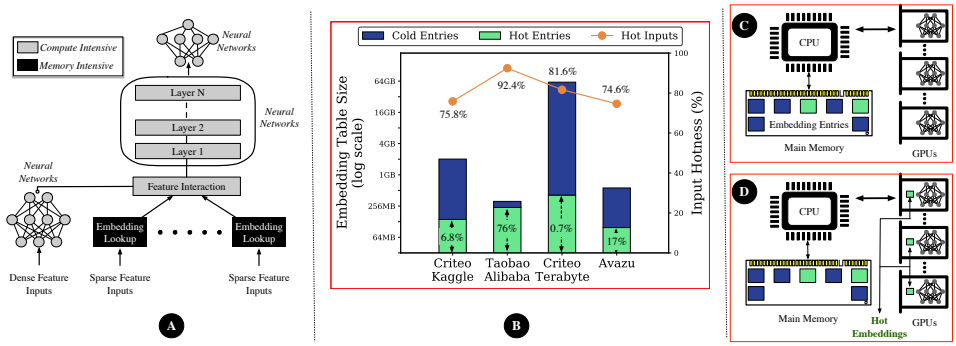


Figure 1.1: A) Typical recommender mode [23, 26, 41]. They comprise compute-intensive neural networks like DNNs and MLPs in tandem with the memory-intensive embedding tables. B) shows embedding table sizes for four real world datasets and the proportion of the embedding table that is frequently accessed (hot). The graph also shows the % of training inputs that only access the hot embeddings. C) shows the baseline embedding data layout, i.e., storing entirely in the main memory. D) shows the proposed layout where hot embeddings that cater to over 70% of the training inputs, are stored locally on GPUs.

1.3 Approach and Contributions

This thesis focuses on developing a framework to train deep learning recommendation models (DLRM) by proposing the Frequently Accessed Embeddings (FAE) framework that efficiently places embedding data across CPUs and GPUs. while maintaining baseline accuracy.

Contributions: This paper makes the following contributions:

- We find that embedding table accesses in real world recommender models is heavily skewed, thus allocating equal compute resources to all the entries is sub-optimal.
- We intelligently place hot embeddings on every GPU device involved in training while retaining cold entries on CPUs. Placing only hot embeddings on GPUs reduces its memory requirement and improves performance. This is because FAE eliminates CPU-GPU communication for inputs that access hot embeddings and enables accelerating the computation of the model for frequently accessed embedding entries.

- To optimize training, FAE performs sampling of the input dataset to determine the access pattern of embedding tables. Thereafter, FAE classifies the input data into hot and cold categories. FAE ensures that a minibatch either accesses only hot or only cold embeddings to avoid communication overheads. At runtime, FAE intertwines executions of hot and cold input mini-batches to ensure the baseline accuracy.

- FAE employs statistical techniques to avoid traversing through the entire input dataset and embedding tables to determine the hot embedding access threshold and the size of the hot embedding table while incurring negligible overhead.

We prototype FAE on well established open-source deep learning-based recommender system training models DLRM [41] and TBSM [26]. These models are adopted by both academia [11] and industry [19, 44, 60]. We compare our FAE optimized training with two implementations. First, the open-source implementations of DLRM and TBSM. Second, a highly optimized implementation of these models using the XDL framework [30]. We choose the commercialized industrial deep learning framework, XDL as of high utilization of CPU. This deep learning framework for high-dimensional sparse data, utilizes CPUs and offers a notably scalable solution for training recommendation models. Hence, it's been chosen as well-established state-out-the-art baseline to compare our model with.

We evaluate FAE for a wide variety of real-world and synthetic deep learning based recommender models. For real-world model architectures, our experiments show that FAE achieves, on average, a performance improvement of 2.3× and 1.52× in comparison to XDL enhanced CPU and CPU-GPU baseline, respectively. Furthermore, FAE achieves 4.76× and 1.80× against the open-source implementation of DLRM and TBSM on CPU and CPU-GPU, respectively. Both baselines execute in a hybrid mode that uses a CPU with 4 GPUs. FAE reduces the amount of data transferred from CPU to GPU by 1.54× in comparison to XDL-based baseline. For synthetic model architectures, FAE achieves 2.94× speedup over XDL-based baseline.

The rest of the thesis is organized as follows. We first give a brief background identifying the problem in Chapter 2, then we provide a brief overview of related

work in Chapter 3. We identify the main challenges we overcame in Chapter 4 and present our approach for handling them in Chapter 5. We evaluate our proposed techniques in Chapter 6. Finally, we conclude the thesis and discuss avenues of future research in Chapter 7.

Chapter 2

Background

In this section, we first present a brief primer on architecture of recommendation systems and some industrial approaches to train them, then define the terms we use, followed by our training process.

2.1 Recommendation models and their training inputs

2.1 shows the flow of a recommendation model which comprises embedding lookup and neural network layers. The recommendation model has two types of inputs, sparse and dense feature inputs. Sparse inputs typically denote specific preferences of the user (like the movie genre, choice of music, etc.) and are used by the embedding layers. Dense inputs are representation of continuous properties of user or item such as time of day, location of users, etc. that feed directly into the neural network layers.

The embedding phase uses large tables containing data that reduces the sparse input feature space into a vector. These inputs are used by the deep neural network (DNN) and multilayer perceptron (MLP) components to classify and determine the final recommendation.

2.2 State-of-the-art mode of execution for training

Machine learning techniques generally employ data-parallel training to reduce the overall execution time. This mode of training requires model replication across all

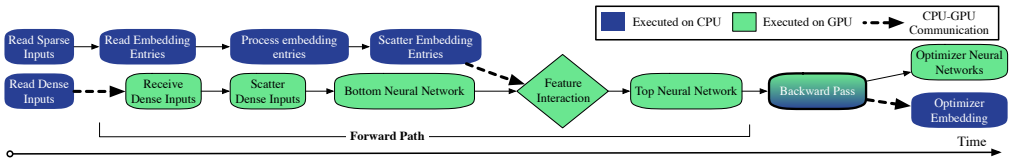


Figure 2.1: Execution graph of deep learning based recommender model. In this graph we show the forward graph in detail, the backward pass is a mirror of forward and executes on CPU and GPU according to its forward counterpart. The current mode of training for DLRM and TBSM requires embedding storage, reading, and processing, on CPU.

the GPU devices, where each device executes on different inputs in a mini-batch. Thereafter, a post-execution synchronization is performed to update the weights/parameters using the aggregated gradient values. For recommendation models, this training mode tends to be infeasible as embedding tables cannot fit even on high-end GPUs such as Nvidia-V100.

To overcome this issue, as shown in the 2.1, past work either executes the whole graph on the CPU or uses the CPU to handle the memory-intensive embedding layer with the GPUs executing the compute-intensive DNN layers. The first case is inefficient as CPUs are not optimized for neural network training as they can not optimally process large tensor operations. On the other hand, the hybrid CPU-GPU mode incurs CPU-GPU communication overheads for intermediate results and gradients. This is shown in the forward pass by the bold dotted lines in the 2.1. The backward pass also executes in a CPU-GPU mode, with CPU executing the backward computation for embeddings and GPU executing the backward propagation of neural layers. Thereafter, the gradients are generated on CPU for embeddings and on GPU for neural layers. Our experiments show that CPU-GPU communication can take up significant percentage of the total training time. Additionally, any computation involving embedding data, such as the massively-parallel stochastic gradient descent (SGD) optimization, also then executes on the CPU.

2.3 Leveraging training input and embedding access patterns

Data accesses can exhibit certain locality that can be exploited either at software, system, or hardware level. For recommender models trained on real-world data, some sparse inputs are significantly more popular than others. Therefore, in such real-world applications, accesses into embedding tables are also heavily skewed. For instance, for the Criteo Kaggle dataset on DLRM, the top 6.8% of the embedding table entries observe at least 76% of the total accesses.

It is important to note that the cold portion of the embedding data is critical from a learning perspective as it contributes towards the overall efficacy of the model. This is because cold embeddings help cater the model to a wider user base. Thus, training only on popular inputs would make the targeted user experience futile as it would lead to certain items being always recommended. Nevertheless, from a memory and infrastructure perspective, as shown in Figure 1.1, hot entries are more important as they from 75% to 92% of the total training input accesses.

In this thesis, the proposed framework leverages the popularity semantics of training input to mitigate the bottlenecks of the above mentioned CPU-GPU execution by optimizing the embedding data layout in the memory hierarchy. Intuitively, highly accessed embeddings are kept in close proximity to the compute, i.e. GPU, whereas the cold embedding entries are stored in relatively larger but slower CPU memories. This allows us to execute the entire training graph, shown in 2.1, on the GPU in a data-parallel fashion for the popular inputs. This data layout overcomes limitations of the baseline by:

- Accelerating embedding compute through GPUs whilst being within the memory capacity of the devices like CPU.
- Eliminating the communication overheads (gradients and activations) between CPU and GPU.

Chapter 3

Related Work

Training machine learning models is an important and heavily developed area of research. Optimizing training for deep neural networks [7, 25, 28, 40, 47] has garnered most of the attention, whereas recommender models have been under-investigated. We classify related works into four broad categories, delving into motivations and prospective benefits for each. The first category includes works related to optimizing data through caching using reinforcement learning, ensembling of multilayer perceptrons, sequential recommendation system, and hardware optimizations. In one of the subsections, we discuss a paper that proposes compression techniques for web-scale recommendation system in order to improve the scalability and reduce memory footprint. Then, we go over the optimization to reduce the memory for sequential recommendation model. At the end, we go through the papers regarding near-memory processing. The second category summarizes approaches that exploit GPUs to run the model by tweaking and training the DNN on GPU. these category of approaches unleashes the potential parallelization with faster weight updating. Getting into more details, the main problem is that with respect to DLRM memory footprints, they are too large to fit on devices like GPU. Hence, scalability is the major issue for these class of solutions. In this section we go over papers proposing an approach to improve scalability. The third category goes over the fact that compressing the whole model can eliminate such as communication and bandwidth overheads. The main issue is embedding table placement as of large size of embedding tables. In this section we through works done to

overcome this issue by various solutions such as hierarchical memory storage to eliminate or mitigate the CPU-GPU communication overhead.Finally, the fourth category goes over distributed training for recommendation models.

3.1 Optimizations data layout through caching

Works in the past [54, 57] have delved into informed and domain-aware caching with their ever increasing requirement for compute and memory. In the deep learning realm, some of the prior works like Quiver [35] cache data on local SSDs to eliminate slow reads from remote storage. Furthermore, they employs hashing based techniques to incorporate thrashing-free strategies across jobs to efficiently utilize the shared cache.

Instead, this work dives into the semantics of the training inputs observed by recommender models and offers compile time strategies to statistically ensure hot data is placed close to compute. Generally speaking, using conventional caching technique will not give the model any significant benefit as twofold that they function based on. Temporal and Spatial locality.

3.1.1 Cache-based Optimization

Works on caching mostly have been focused on caching the embedding table entries. The issue arises as of random access pattern. At the stage of embedding lookup, we fetch the required entries based on the input and the model and do operations such as aggregation. We then, come up with a embedding table vector as a result send to the next stage or feature interaction. As of this unpredictable and random access of embedding entries, it might not be feasible to effectively cache. Hence, it leaves the possibility of effective caching to the dataset instead of the scheme and model. This behavior depends on entity's property (item or user) and makes it an unappealing approach to optimizing the data layout if all inputs are treated the same during caching. Due to aforementioned factor, an intelligent mechanism for caching is required which can be carries out by extracting data pattern. The more popular and more frequent the embedding entry is, the higher the odds of storing it in the cache should be.

Our offline techniques enable FAE to fully exploit coarse grained GPU based

compute throughput, without employing any dynamic hashing. The prior work ATML [22] has been exploiting the fact that not all the embedding entries are accessed equally. Some are requested to get fetched more frequent than other ones depending on the input. This pattern can be significantly beneficial for compressing the model. Hence, caching will more effective by fitting more embedding entries. Moreover, it reduces the amount of bandwidth required.

An embedding table entry is comprised of multiple dimension. ATML exploits the insight that not all the sparse dimension are effecting the performance of the model the same, thus some can be masked off. It proposes a twined-based layer scheme called Adaptively-Masked Twins-based Layer (AMTL). ATML uses two branches of most frequently used sparse dimension and the least frequently used embedding entries to train a neural network model. The neural network determines which sparse dimensions should be masked off. The insight behind this scheme is that is least frequent dimension will have their parameters less updated and overshadowed by more frequent dimensions if parameters are not separated. This flexible method of determining the entry size has been proposed to improve the fixed size embedding table to unleash higher degree of compression and cache improvement. As by profiling the input, more frequent embedding entries will be determined and cache replacement policies can perform more efficiently.

3.1.2 Tensorization

Based on experiments, more than 99% of memory footprint of DRLM model belongs to the extremely large embedding tables which are in order of GB or even TB. The trend of large memory size is expected to get exacerbated in the future. The underlying logic is that with items and users are added to the model, the capability of model to suggest more fine-tuned recommendation increases and that requires higher cardinality and higher embedding dimension for embedding tables. On the other hand, this aggravates the memory-intensity problem in addition to higher bandwidth required to train the model.

Work in [4], for instance, assumes that the major part of memory footprint for a neural network comes from the weights of fully connected layers and that can be transformed into Tensor Train or TT-format using an algorithm called tensor

value decomposition using a format called TT-layer. Tensor value decomposition represents the same network with less number of parameters using TT-format. This work doesn't require any change of training algorithm while being able to maintain the baseline accuracy.

Work in [46] on the other hand, uses the same representation for a different context. It applies the same approach to reduce the size of embedding layer although this comes at the cost of negligible drop of accuracy. In this approach, first a tensor has been reshaped and then, it will be decomposed to multiple lower-ranked tensors called TT-Cores. These cores are the same as trainable parameters for the embedding layer and multiplication of them all results in the original tensor.

Work in [11] takes one step further and applies the tensor decomposition method to the memory and bandwidth-intensive part of DRLM or embedding tables. TT-rec decomposes some of embedding tables into lower rank tables called TT-core. Product of multiplication of TT-cores results in the original embedding table. Without any further optimization, TT-rec loses some accuracy. To compensate for that, it deploys a method to initialize the weights of embedding tables. Another overhead incur by this approach is increase in training time as of decompression. To mitigate it, TT-rec saves the more popular and frequently used embedding table entries closer to the computation so it will take less time to access them. It does so by storing them in a cache in a uncompressed manner. This also helps to revive the lost accuracy as well and reach the baseline.

TT-Rec is suitable for devices with high computation-to-communication ratio like GPU. As large embedding tables are the major part of memory footprints of DLRM, TT-Rec decomposes this component to first incur the least overhead second gets the most of memory footprint reduction.

3.1.3 Reinforcement Learning based

Similar to ATML [22], The paper ESPAN [21] follows the same incentive but with a different scheme and approach. This work deploys reinforcement learning techniques to figure out the the best size for each embedding table entry in an automative manner. ESPAN uses the insight that low-dimensional embedding entries can get trained swiftly with less frequent data in contrary to the more frequent ones.

It is probable that they overfit if we shrink the feature size the same. Therefore, the cardinality of the embedding entry should be proportional to their frequency of use. Hence a unified length for embedding entries limits the recommendation model performance. In addition to that, The frequency of both user and item embedding table entries change during runtime. This necessitate a dynamic scheme to figure out the proper dimension size with respect to time accordingly.

ESPAN exploits The aforementioned twofold insights to compress the memory using an automated reinforcement learning agent. It picks a discreet search space of acceptable integer numbers for the length of a embedding vector. These number are usually a power of two.

The scheme uses streamed processing of embedding table entries either for user or item. ESPAN is comprised of two main components. The first part is the Deep recommendation model and the second one is policy network that operates as a reinforcement learning agent that determines the length of embedding table entries. This network exploits frequency and current length of the input feature vector, maps the integer frequency to a numerical embedding representation. It then, glues them to each other, passes them through a multilayer perceptron to decide the prospective size. The intermediate product is consumed by a softmax function to make a binary classification.

After whole process, a feature vector either increase in size or remains the same. The reason is that that using streaming input, the popularity of an feature vector increases or stays the same. Hence, there are two classes associated with class prediction.

After applying the policy network, if the decision is enlarging the vector, it uses a linear transformation to fetch the larger chosen feature vector from the embedding. In this way, we don't have to initialize the embedding tables from scratch and can use the updated embedding table up to the point. At this point, both vectors are consumed by the Deep recommendation model to make a prediction and it will be passed to the Reward function. The reward function then calculates the loss and do backpropagation for the weights of multilayer perceptron in the policy network.

3.1.4 Web-scale Recommendation Model

Some orthogonal research has been conducted regarding the scalability of compression schemes for specific application for instance web-scale recommendation. For these variation of applications, the deployed compression scheme should be scalable yet timing needs to be taken care of. In general, timing, each work incurs should not be extensively significant. The overhead of aforementioned schemes for instance [21, 22] so far is correlated with the number of rows in the uncompressed embedding table entry. This makes these scheme undesirable to be used for web-scale recommendations as of lack of sufficient scalability.

The paper Binary Hash [6] addresses this issue by proposing a novel binary code based hashing scheme to compress the embedding tables as the memory-intensive component of the recommendation system, without jeopardizing the performance of the model significantly. This paper achieves 1000x reduction in model size yet preserving 99% of the baseline performance. Binary Hash stays away from deploying the conventional modulo-based hashing function. As they comes at the cost of high rate of collision. Using these hashing functions, many rows in the embedding tables are mapped to the same place and that makes the compression lossy and higher overhead in comparison to the cases without collision.

Binary Hash is comprised of mainly three steps. Given that input data for recommendation models (categorical feature values) can be including both integer and string, the proposed framework starts off with feature mapping of both of these data types. In the beginning, the raw features would be mapped to unique 64-bit integer called Hash-id using hash functions like Murmur Hash [16]. In the next stage we convert the Hash-id into binary. Then, we divide the final results into separate blocks. This is what we refer to as binary code hashing. Each of those blocks represent an unique index to an embedding table entry. To reach an efficient code blocking, Binary Hash proposes two chief strategies, succession and skip. The former pairs up the successive bits in the same block. On the contrary, the skip strategy, puts the bits that are some interval apart depending on the degree of skipping. For the further stages in the recommendation like embedding lookup, the paper uses fusion to append the embedding table entries out of the code blocks. The appending function varies from LSTM, concatination or pooling.

3.1.5 Sequential Recommendation Model

Recommendation systems can fit into two categories. The first one uses the matrix factorization approach to extract the pattern of items each user has selected based on previous interactions. On the other hand, Markov chain-based or sequential models, maps the previous interactions to a graph trying to model the sequential behavior. This aforementioned type of recommendation system can be used in online advertising or e-commerce. The paper [62] bridges the gap between Deep neural network and their application in recommendation systems. It addresses the issue of large memory overhead by proposing a temporal context aware embedding composition using light-weight attention network called LSAN. LSAN aims to learn the short-term and long-term pattern of users behavior.

LSAN comes up with a couple of base embedding vectors and then tends to form other ones based on the fusion of the base vectors using quotient remainder strategy. The reason is that this strategy doesn't incur any more learnable parameter hence significant overhead to the framework. The main novelty of this papers lies in the fact that using multi-head attention, layers can actually improve figuring out the local and general signal for user preference. To obtain the local signal, LSAN exploit convolution. On the other hand, for the global signal, it uses multi-head attention layers.

3.1.6 Hardware Optimization

Papers such as [38, 64] employ runtime techniques to improve memory, communication, and I/O resources for training and reduce data stall time, respectively. The framework proposed [64] is called DeepIO. DeepIO has been proposed to find a reasonable balance between IO, memory and communication resources to train the large deep neural network efficiently. Recently, as neural networks are getting deeper we need larger datasets to train them. These datasets should be loaded from a memory to form different batches. The batches should be shuffled to avoid overfitting and this shuffling incurs a lot of random accesses to the memory and that hinders the training overall. Likewise, the embedding training component of DLRM model exhibits the same issue. Due random access pattern of embedding table entries, the same insight from the paper can be exploited. As training this

large models can be challenging due to low reading speed and being memory intensive. DeepIO resolves this issue by techniques such as input pipelining and entropy-aware opportunistic ordering.

Input pipelining technique has been implemented at two different levels. One is associated with reading the dataset from the disk and has been designed to mitigate the significant overhead of it. After that, data has been transferred from disk to another buffer as part of DeepIO as an intermediate level of memory. This hybrid backend-memory pipeling amortizes the effect of random access pattern to form a minibatch and the high latency to fetch data from disk. Likewise in DLRM, DeepIO can be used to mitigate the memory-intensity of embedding table as first they are large in size (recent estimation based on Facebook models is 100s of GB) so they have to be originally stored on the disk.

Secondly, due to random access pattern of embedding tables entries, there is significant overhead of fetching required entries as it hinders exploiting spatial and temporal locality. Making the best of locality is conventional approach to mitigate the memory overhead by taking the reusability into account.

On the hardware side, prior work [14] proposes an technique called Bandana to store embedding tables in non-volatile memories and allocate a certain portion of DRAM for caching. Unlike DRAM with lower capacity but higher bandwidth, non-volatile memory (NVM) benefits from large size. On the other hand embedding table component of recommendation model is so large that makes the operations infeasible. That can make NVM a likely option to store the whole embedding tables and use DRAM as last-level cache (LLC). However, NVM suffers from low bandwidth. Bandana's main goal is to exploit the DRAM's bandwidth as much as possible by placing the embedding table entries that are most likely to be fetched together. Besides, it determines which embedding table entries should be cached and stored there.

This work, however, does not support GPU based training executions with replicated hot embeddings and does not deal with perceptive input preprocessing to reduce the overhead of communication between devices.

3.1.7 Near-memory Processing

Recent works [17, 33, 51] have also proposed solutions to accelerate near-memory processing for embedding tables, but do not facilitate distributed training of entire recommender models using GPUs.

Mixed Dimension [17] aims to exploit the skew within input dataset for recommender systems as there are based on human's choice. For instance, intuitively, if 100 movies rolled out at the same time, a few will be are more likely to become blockbuster. Therefore, skew is pretty much likely to exist in all the recommednation systems dataset. Given this intuition, Mixed Dimension, modifies the length of popular embedding entries. The popular ones become larger whereas the rest shrink in size so that the total memory budget stays the same.

On the other note, RecNMP [33] aims to speed up the performance of recommendation system by exploiting the insight that production-scale recommendation model are restricted by memory bandwidth for inference mostly as of large embedding tables. In this regard, the paper proposes RecNMP to place the data closer to the memory using a DRAM compliant. RecNMP operates based on near memory processing to reduce the amount of bandwidth required.

It designs targets the fact that different rank of the memory can fetch the data in parallel. Benefiting from this, it designs a few intermediate modules between the memory and the memory controller. These modules are comprised of light-weight computation units for lookup operation, which is the main bottleneck here, and a cache structure. Based on the close analysis of the RecNMP, Although embedding table entries doesn't follow pattern in term of spatial locality, exploiting temporal locality is completely feasible. This is exactly equivalent to the observation for the popularity and means that some embedding table entries or mapping of some items or users, are more likely to be accessed in comparison to the rest.

The cache employed in the intermediate memory module is responsible to capture those popular entries so that bandwidth can be efficiently exploited and that reduces the congestion arised from inserting the embedding table entries closer to the CPU cache along with other data. Although, the paper achieves a acceptable speedup for inference but our paper aims to improve the training performance which has different bottleneck rather than the one targeted by RecNMP.

[51] takes a different approach. This paper tackles the memory capacity problem. As we mentioned the capacity bottleneck stems from the large size of embedding tables. This paper claims to overcome this issue by reducing the size of embedding tables using hashing. Conventional hashing proved to be not efficient enough to reduce the size of embedding tables as of significant rate of collision. Hence, CompEMB [51] shrinks the size of the embedding tables by dividing them with various methods such as quotient-remainder. The size of both combined will be significantly less than the original embedding tables. Moreover, this method can be done recursively meaning the quotient matrix can be broken down into smaller embedding tables and so on.

This partitioning will be complementary meaning that not even a single embedding table entry will be replicated in two different partitions. This approach incurs extra memory needed to store the metadata given that the same index can refer to different embedding table entries in different partitions. On that note, after acquiring the entries required to perform the embedding table lookup, the paper proposes to use non-conventional operations such as MLP or element-wise multiplication as a substitution of the aggregation for baseline.

This can result into representing different various combination of embedding entries swiftly but the drawback is that it eliminates the intrinsic aggregation properties such as associative and commutative. For instance, let's say after getting all the entries required, the model has been designed in such a way that a MLP produces the output.

We know for a fact that MLP does not exhibit the aforementioned properties of aggregation as of non-linearity. Generally, CompEMB aims to reduce the number of parameters and maintain the baseline accuracy. On the other hand, [51] does this by dividing the model into complementary components but this might backfire if the intrinsic characteristics are not captured in the produced components. Additionally, this arises some additional challenges depending on the dataset. If not captured properly, this approach can result in performance reduction.

3.2 Efficient execution of tasks on GPUs

There has been a wide variety [36, 59] of works across domain of optimizing GPU execution by improving throughput and utilization. HippogriffDB [36] for instance, addresses the contrast between the fact that embedding tables are becoming larger every year. In addition to that memory on GPU is limited. Therefore, we can not fit the whole model on one GPU to accelerate the training or inference process to properly scale the model.

Given that, this paper proposes HippogriffDB an efficient and scalable system that comes up with a datapath by compressing the model and aiming for more compression. It stores the model in the compressed fashion on the memory and decompresses it on the GPU to save the bandwidth and improves the scalability. Also, it benefits from a novel data transfer that directly sends the data from SSD to GPU using NVM (non-volatile memory). This can be significantly beneficial for storing embedding tables in the case that system uses any NVM as a larger memory hierarchy. These type of memories can be used as an alternative instead of distributed training. In general model is too large to be stored on a central server to be dealt with given that these large models needs to be stored in databases.

HippogriffDB performs query-over-block execution to unleash the GPU computation power and deal with the data that is larger than the size of the GPU memory. In this regard, it uses two buffers to accelerate the operating in addition to operation fusion mechanism.

As mentioned, as of the large size of embedding tables, they can be stored in larger memories such as NVM or DRAM. These memory hierarchies can play the role of database that store the memory-intensive component of recommendation models. Hence, we need to fetch the required entries as queries to send them to GPU to be handled. Based on Concurrent Queries [59] observation, GPUs can not be shared while taking care of concurrent queries. This results in underutilization of these computation resources.

Concurrent Queries targets this problem by running queries concurrently on GPU to improve the system throughput. This paper introduces MultiQx-GPU as the framework to run multiple queries at the same time on GPUs. MultiQx-GPU aims to follow two main points. First versatility meaning that it attempts to GPU

model-agnostic approach so that regardless of the model, the implementation will be feasible. Second is high efficiency. GPUs have vast applications such as gaming etc. Therefore, the idea of execution of multiple queries at the same time aligns with the purpose of using GPU in the first place.

The major problem comes from the fact the GPUs don't have the software layer to context switch or deal with resource contention. Currently, unlike CPU, there is no such concept or design that handles fine-grain context or abstraction of virtual memory on GPUs. In this way Concurrent Queries designs a scheduler and a novel scheme to swap the contexts from the memory.

MultiQx-GPU provides the solution in three different layers. First, exploiting a query scheduler similar to the one that operating system grants CPU along with device memory manager same as the virtual memory table. In the layer of interfacing with GPU, MultiQx-GPU uses libraries like CUDA or OpenCL. MultiQx-GPU perform query executions by more efficiently by merging jobs into groups, thus reducing the setup overheads and increasing the throughput.

3.3 Embedding parameter placement

Works in [63] offers a hierarchical parameter server that builds a distributed hash table across multiple GPUs. This work stores the working parameters close to computation, i.e, GPU, at runtime, albeit treats all embedding entries equally. Instead, FAE delves into the access pattern of each dataset and uses this information to store the highly accessed embedding entries in the GPU for the entirety of the training job.

As baseline Distributed hierarchical parameter server [63] uses the MPI-cluster model. In this approach the entire model and the batches are distributed across all nodes to get fully parallel. During backward propagation for training phase, the nodes pull the data required from other nodes using MPI primitives. The drawback of this approach is the high cost of communication for transferring all the data required. In this work, the paper eliminates the CPU-GPU communication as much as possible by proposing the hierarchical memories for storing the parameters. Also, it uses hashing method to reduce the model footprint.

Distributed hierarchical parameter server comes up with a pipeline all the way

from SSD to GPU. As of the large size of the target model (10 TB), it uses SSD to store the entire model. At each stage of the pipeline model will be sharded on parameter servers. Each of these servers are responsible for distributing the model and batch on a set of workers or GPUs. After the GPUs are done with the back propagation, then the updated parameters will be send to the second stage or memory parameter server and then to the SSD where we store the entire model.

Prior work in [3] aims to address intricacies of using GPU to train recommendation models. The main problem arises as the critical path of deep learning based recommendation system is comprised of two major components of deep neural network which are light-weighted and computation-intensive in contradiction to large memory-intensive embedding tables. It aims to understand the implications of different embedding table placements within an heterogeneous data-centre. However, none of the techniques leverage runtime access skew for their embedding table placement that can improve the overall training performance.

Work in XDL [30], targets training of recommendation models with high-dimensional sparse inputs. This Framework is comprised of two main components of SparseNet and DenseNet. The latter is composed of multiple dense layers and is notably computation-intensive. The SparseNet, on the other hand, is composed of millions of features out of raw input samples. XDL exploits approach of distributed training of deep learning based models. Each worker can only be CPU, whereas, AMS can be composed of both CPU or GPU. XDL deploys a server called Advanced model server (AMS) to save the model on. Another component is Back-end worker that can store any deep learning model such as Deep Neural Network (DNN), attention mechanisms, etc.

During forward path, each back-end worker acquires the inputs from I.O. stream. In the next step, each of these back-end workers send the feature ID to the AMS. AMS performs the lookup operation and send the results back to each worker as well as the last updated version of the model. Workers move on with passing the dense input that has been received from the AMS. Passing through the DenseNet, it outputs the result. The reverse sequence will be taken during backward path and gradients will be computed by workers. AMS will gather all the gradients both for SparseNet and DenseNet. Parameter updating and optimizer operation will be done on the server side and the lastest version will be kept there.

3.4 Mitigating memory intensive training through compression, sparsity, and quantization

Past works have used compression [27, 55, 61], sparsity [15], and quantization [20] to reduce memory footprint of machine learning models. For instance, prior work Gist [27] has been exploring ways to compress deep neural network models. The problem arises as with more complicated applications the need to have deeper neural network to overcome overfitting. However, these deep models can not fit into the current conventional GPU memories. This results in numerous challenges. First, we need to distribute the model on multiple GPUs and that essentially incurs higher overhead in comparison to the uni-GPU approach hence, limits the size of the neural network that can be trained. It exploits from insight that the main memory footprint comes from the intermediate feature maps and these layers will be accessed twice during training. Once during forward and the backward. This shows that feature maps can be reused as of the temporal reusability. Gist comes up with layer-specific encoding scheme in two forms of lossy and lossless. In addition to that, Gist delves into the characteristics of neural network to compress the values. Having said that, in DLRM, the same problem arises from the capacity and bandwidth-intensive component or embedding table. This makes a significant difference as first random access pattern dominates the indices accessing the embedding table entries. On the other hand, temporal reusability is guaranteed for the neural network weights. Whereas for embedding table lookup operation, mostly aggregation will be applied and no spatial locality can be assumed along with fact that even temporal locality might not be guaranteed.

Prior work in[45], optimizes training by modifying the model either through mixed-precision training or eliminating rare categorical variables to reduce the embedding table size. This reduction can help fit the model on devices like GPU with a fairly low memory budget in comparison to CPU but efficient in terms of reaching a higher throughput for deep neural network. Therefore, aiming for reducing the size, not only do we overcome the capacity problem but also we tackled the communication and bandwidth problem.

Originally in the DRLM model by Facebook, the bottleneck for forward path was the capacity as of large embedding tables. This problem can be mitigated once

the model has been reduced. In addition, one of the bottleneck is butterfly shuffle during backward path, embedding tables are sharded and then placed on GPUs. Hence, to operate the embedding lookup stage, GPUs need to send and receive the associated entries that might be present on other GPUs. Other GPUs do the same thing. This operation requires quite a robust and a fast link between GPUs like NVL to mitigate the send and receive requests. The same goes for pipelining. As of implemention, sending and receiving during the backward path can be interleaved. None of these arises if the entire model can fit on just one device.

But even with these optimizations real dataset's entire embedding table cannot fit on a GPU. Moreover, approaches that change the data representation and/or embedding tables, require accuracy re-validation across a variety of models and datasets. enables apropos utilization of memory hierarchy without employing overheads such as compression/decompression in HippogriffDB [36] and sparse operations. moreover, performs full-precision training of the baseline model by leveraging the highly skewed access pattern for embedded tables and increase the throughput for hot embedding entries. Nevertheless, our framework is orthogonal to the prior techniques and can be used in tandem with them to improve the memory efficiency even further.

3.5 Distributed deep learning training

Data parallel training [34] forms the most common form of distributed training as it only requires synchronization after the gradients generated in backward pass of training. As models become significantly larger [50, 52], model parallelism [9, 25] and pipeline parallelism [40] are becoming common as they split a single model onto multiple devices. The same goes for DLRM as well. Not only is this model large enough that not all components can be fit on the GPU memory entirely but also the critical path is comprised of both computation-intensive such as neural networks and memory-intensive part such as embedding table. Hence, to tackle this challenge, we deploy both model and data parallel technique to train the model in the distributed fashion.

Work in Pipedream [40] is taking it one step further. Conventionally to train DNNs, intra-batch pipelining is performed meaning that worker trains the model

using its portion of data and then synchronizes the weights of the model with other workers using primitives like AllReduce. Not only is this time-consuming but also communication might exceed computation time. In order to mitigate it, PipeDream interleaves backward path of batches been taken care of prior in time with forward path of latter batches. It also does the model partitioning on various workers based on layers. During backward for training DNNs, layers have been dispatched off to different worker. For instance, it's determined that input layer has been placed on worker number one and first feature map layer has been placed on device two. Therefore, during forward,device one sends information to device two. On the other hand, on backward path device two will send information to device one. In DLRM, on the contrary, every worker needs to not only distribute the part of model present on the device to all the other devices but also during backpropagation.

PipeDream does the pipeling to not only reduce the number of times communication occurs but also reducing each communication by overlapping it with computation. Nonetheless, the techniques employed to automatically split the models [29, 56], offer model parallelism solutions to enable training of large model with size constrained by the accelerator memory capacity. However, none of these techniques dive into the semantics of input data to perform an optimal split. This is because they are mainly suitable for DNNs.

3.6 Summary

In this section we covered the prior works regarding training recommendation models and how thye target an efficient end-to-end model training. In the first section, we delve into approaches to optimize the data layout such as cache based optimization, reinforcement learning based, web-scale recommendation model, sequential recommendation model, hardware optimization and near-memory processing. Next, we go through prior works regarding the efficient execution of tasks on GPU. In the next section, different strategies for embedding parameter placement has been added. In the next subsection, strategies to reduce the footprint of the model such as compression, sparsity and quantization has been discussed. In the end, prior works regarding the distributed training has been discussed.

Chapter 4

Challenges

To perform efficient end-to-end training with the optimized embedding layout while maintaining baseline accuracy, we require a comprehensive framework that has both static and runtime components. Next we analyze the challenges of such a training execution. Overall, there are four challenges being discussed in this chapter. First, how to maintain the baseline accuracy while changing the data layout. Second, how to efficiently segregate both both hot embeddings from cold one as well as segregating hot inputs from cold inputs and store them in different FAE format to form the cold and hot minibatch from respectively. Third, we uproot the necessity for scheduling hot and cold minibatches and the proper rate. Finally, we discuss the mechanism to maintain the consistency between the embedding copies between devices.

4.1 Maintain Accuracy While Moving Hot Data to GPU

As shown below 4.1, even if 99% of the inputs are popular, i.e., access hot embeddings, the probability that the entire mini-batch accesses *only* hot embeddings decreases dramatically as the minibatch size increases. Hence, it is likely that at least one input within a large minibatch requires accessing cold embedding entries. To obtain benefits from embedding data layout, we require the entire minibatch to only access hot embedding entries. Even a single input accessing cold embedding entries can stall GPU execution as it incurs CPU-GPU communication.

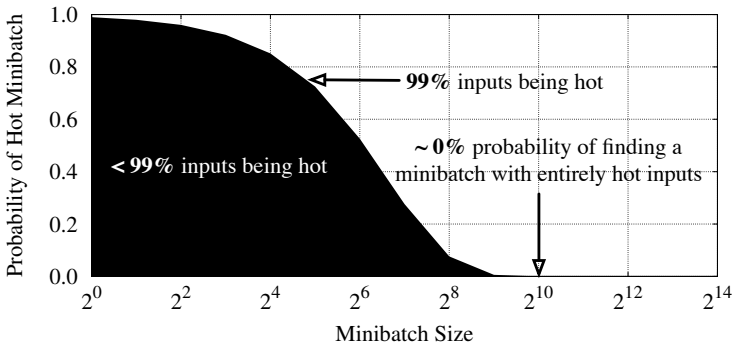


Figure 4.1: Probability of creating a mini-batch with all popular inputs when the number of hot-inputs are 99% or lower. This reduces drastically as the mini-batch size increases.

To overcome this challenge, our framework comprises a static component that performs input-dataset preprocessing and organizes mini-batches such that they completely contain only hot or cold inputs. Each input is consist of indices each embedding table in the model. We define an input hot if and only if for all embedding tables, it accesses hot embedding entries otherwise we consider the input as cold. It is probable for some cold inputs, some of the accesses to some embedding tables are to hot embedding table entries. An hot embedding entry is considered hot if and only if the total number of accesses to it surpasses a certain limit This preprocessing needs to be performed only once per dataset and is stored in a preprocessed format for subsequent executions.

The size of the final format depends on the size of the original dataset. The whole preprocessed FAE-formatted dataset will be stored on CPU memory as well as the cold and hot minibatches. For hot mini-batches, the framework performs GPU-only data-parallel execution and for cold mini-batches the framework falls back onto the CPU-GPU hybrid execution mode.

4.2 Cold and hot embedding table entries segregation

The classification of an embedding entry as hot or cold is based on the access threshold. Any entry that is accessed more than a threshold is classified as hot. We expose this threshold as a knob to FAE to adjust the amount of hot embeddings that can be managed by GPUs, based on both the model and system specifications. To minimize performance overhead, we devise statistical techniques that use input

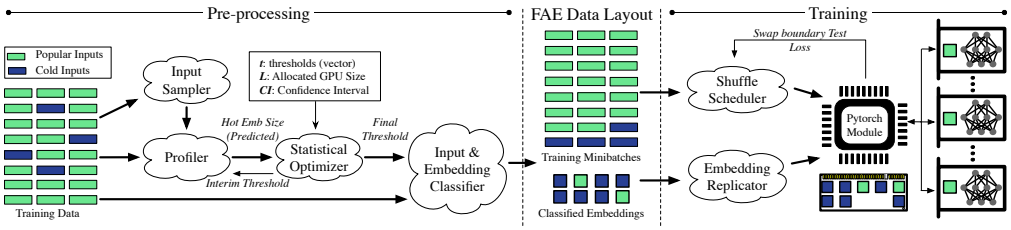


Figure 4.2: The FAE framework. The preprocessing phase calculates the threshold for classifying hot embeddings. This phase uses random-sampling of input datasets and embedding tables to determine the best threshold for hot embeddings. This threshold is also used to classify inputs into hot and cold minibatches. At runtime, GPUs execute the hot input mini-batch while cold inputs execute in a CPU-GPU hybrid mode. The scheduler uses feedback from the Pytorch modules to determine the rate of hot and cold mini-batches swap.

dataset sampling to determine the access threshold. This enables FAE to determine the optimal threshold without scanning the entire training data. FAE selects a threshold that classifies enough embedding entries as hot so that they fits in allocated GPU device memory.

4.3 Scheduling hot and cold minibatches

FAE processed data contains minibatches that are either entirely hot or cold. Scheduling all the hot minibatches followed by cold minibatches incurs the least embedding update overhead as the embeddings only have to synchronized between GPU and CPU once after the swap. However, such a technique can can have an non-negligible impact on the accuracy. This is because the hot minibatches only update the hot embedding entries whereas the cold minibatches cover more embedding entries (both hot and cold), albeit sparsely. To tackle this issue, our framework, offers a runtime solution that dynamically tunes the rate of issuing hot and cold mini-batches to ensure that the accuracy metrics are met.

4.4 Maintain consistency between the embedding tables that are scattered across devices

FAE replicates hot embedding tables across all the GPU devices and CPU contains all the embeddings (including hot embeddings). Thus, we need to perform two forms of synchronization during the training - one across all the GPUs after each mini-batch of data parallel execution and once between the cold and hot swap between CPU and GPU. In the former case, hot embeddings are synchronized using the AllReduce collectives over the fast NVlink GPU to GPU interconnect [2]. In the latter case, the synchronization across GPU and CPU between hot and cold minibatches is performed through PCIe transfer between the GPU-CPU devices.

This communication overhead incurred by FAE is accounted for in the final execution latencies. To reduce this overhead, FAE minimizes the transitions between hot and cold minibatches, without compromising baseline accuracy.

4.5 Summary

In this chapter, we analyze the challenges we overcome. we make sure that moving the hot embedding inputs to GPU guarantees the baseline accuracy. As the second challenge, we clarified the criteria based on which we segregate hot input from the cold counterparts. FAE makes sure that threshold for this has been chosen wisely for instance based on memory budget, etc. Another challenge arised from cold and hot input segregation is what scheduling policy to apply for these two types of input. The last challenge that we overcame is how to maintain accuracy while all embedding tables are distributed across multiple devices.

Chapter 5

Approach

In Chapter 4, we delve deep into the architecture of our proposed framework, Frequently accessed Embedding or FAE. This framework accelerates the process of recommender system training. FAE efficiently utilizes the GPU memory and computation to reduce the communication cost of obtaining embedding data.

5.1 FAE overview

5.4 illustrates the flow of the framework; FAE consists of the input and embedding preprocessing stage that determines the hotness of embeddings by sampling the input training data and the training stage that replicates hot embeddings on all the GPUs and schedules hot/cold minibatches to ensure baseline accuracy. The preprocessing phase converges on an access threshold to classify an embedding entry as hot. If overall number of accesses to an embedding table entry exceeds the threshold then that entry is called hot otherwise cold.

This threshold is based on the allocated GPU memory size, confidence interval, and the CPU-GPU bandwidth. Thereafter, based on the final threshold, the Embedding Classifier and Input Classifier categorize both embedding entries and sparse inputs into hot and cold portions. The preprocessing phase executes statically once per training dataset, and stores the preprocessed data in the FAE format for subsequent training runs. At runtime, the Embedding Replicator extracts hot embedding entries and creates embedding bags that are replicated across GPUs. The overall

timing depends on the dataset but is negligible with respect to the baseline baseline training time of the model.

The Shuffle Scheduler dynamically determines the execution order of hot and cold sparse input minibatches across the CPU and GPUs at runtime. Based on accuracy goals, the Shuffle Scheduler interleaves hot and cold minibatch queues to capture the updates to all embedding table entries. More information will be provided in the following section and to help understand the next few subsections, Table 5.1 provides description of the notations for the design variables in FAE.

Table 5.1: List of Notations

Notation	Description
$\boldsymbol{D}$	Training input dataset
$\boldsymbol{t}$	Minimum number of access to classify an entry as hot
$\boldsymbol{T}$	Total number of accesses into an embedding table
$\boldsymbol{L}$	User-specified allocation of GPU memory for hot embeddings
$\boldsymbol{h}$	Maximum number of hot embeddings that fit in $\boldsymbol{L}$
$\boldsymbol{E}_z$	Size of embedding table number $\boldsymbol{z}$
$\boldsymbol{x}$	Sampling rate for inputs (%)
$\widehat{\boldsymbol{D}}$	*Sampled* training input dataset entries
$\boldsymbol{n}$	Number of Sample Chunks from the embedding logger
$\boldsymbol{m}$	Number of entries in each embedding logger chunk (n)
$\boldsymbol{N}$	Total $\boldsymbol{m}$-sized entries in the embedding logger
$\boldsymbol{k}$	For any $\boldsymbol{t} \longrightarrow$ Total accesses into any embedding entry
$\boldsymbol{H}_{zt}$	For any $\boldsymbol{t} \longrightarrow$ Sample adjusted t per (z); minimum accesses to classify hot entries
$\boldsymbol{C}_i$	For any $\boldsymbol{t} \longrightarrow$ Number of entries in the m chunk with accesses more than H_{zt}
$\bar{\boldsymbol{y}}$	For any $\boldsymbol{t} \longrightarrow$ Mean of $\boldsymbol{C}$
$\boldsymbol{s}$	For any $\boldsymbol{t} \longrightarrow$ Standard deviation of $\boldsymbol{C}$
$\boldsymbol{CI}_\beta$	Confidence Interval of $\beta\%$

5.1.1 Calibrating the Access Threshold

The first goal of the preprocessing phase is to pick an access threshold ($\boldsymbol{t}$) for the embedding entries. We denote $\boldsymbol{T}$ the total number of accesses into an embedding table. The accesses per entry for hot embeddings is $\geq \boldsymbol{t} \times \boldsymbol{T}$.

Any input that accesses only hot embeddings is also categorized as hot. Picking a larger $\boldsymbol{t}$ would imply that only a few embedding entries would have enough accesses to be classified as hot. It would lead to only a small percentage of sparse inputs that would execute completely in a GPU execution mode and thus reduce the overall performance benefits.

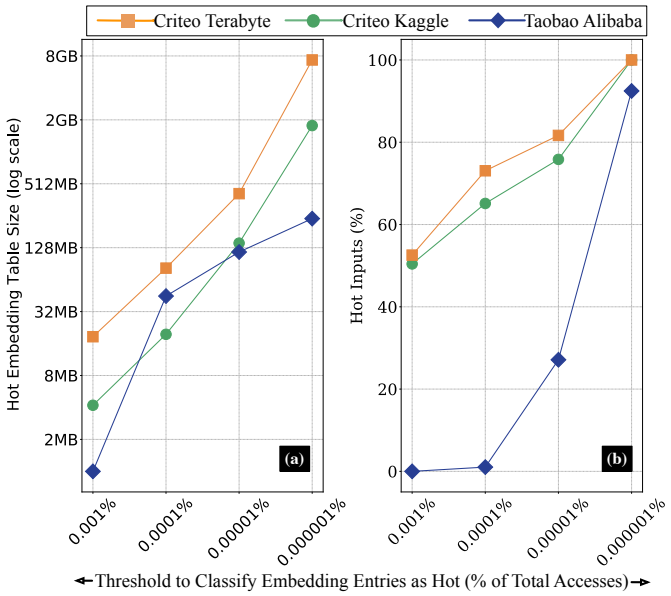


Figure 5.1: (a) Size of hot embedding entries and (b) Percentage of hot inputs with varying access threshold values. As we vary the threshold, the size of the embedding entries increases more rapidly compared to the percent of hot inputs

.

Conversely, picking a small threshold will categorize embedding entries with very few accesses as hot, which would increase the hot embedding table size, often beyond the GPU device memory capacity. Figure 5.1 shows that we observe diminishing returns by reducing the threshold, as the number of hot embedding entries increases more steeply as compared to hot inputs. Thus, we need to efficiently tune $\boldsymbol{t}$ based on the system configuration parameters.

One of the system configuration parameters is the GPU memory allocated for hot embeddings – denoted by $\boldsymbol{L}$. Notation $\boldsymbol{h}$ constitutes the maximum number of hot entries that fit within $\boldsymbol{L}$. A naive mechanism to determine $\boldsymbol{t}$ will profile the *entire* training dataset and analyze the accesses of *all* the embedding entries. This requires sorting all embedding entries based on their access frequencies and classifying the top $\boldsymbol{h}$ entries as hot.

This implementation will incur a high preprocessing overhead as it could imply processing several terabytes of data – even though profiling is performed only once

per dataset.

Instead, we propose a novel *input sampler* and *Statistical Optimizer* that ensures a low static compilation overhead for finding optimal $\boldsymbol{t}$ such that $\boldsymbol{L}$ is used effectively. Figure 5.4 describes the flow of events to determine the optimal value of $\boldsymbol{t}$.

5.1.2 Mitigating Read Overheads with Sparse Input Sampler

As size of the training input dataset is typically very large, we sample $\boldsymbol{x}$% of the input dataset ($\boldsymbol{D}$). The value of $\boldsymbol{x}$ is specified as a hyper-parameter. Our implementation uses $\boldsymbol{x}$ = 5% and obtains $\widehat{\boldsymbol{D}}$ sampled sparse-input entries. Figure 5.7 shows the access profile for one large embedding table each for Criteo Kaggle, Taobao Alibaba, Criteo Terabyte, and Avazu datasets with and without input sampling. Empirically, we observe with a sampling rate of 5%, $\widehat{\boldsymbol{D}}$ maintains a similar access signature as $\boldsymbol{D}$. We scale down the hotness threshold by the sampling rate. Yet, we observe that even with the new scaled-down threshold, hot embedding entries with original threshold, will still be considered hot. Hence, even with scaling down, the access pattern of each embedding table remains the same.

As shown in 5.2, FAE obtains 19× to 55× reduction in latency by input sampling. For the Taobao Alibaba dataset, each input consists of a stream of up to 21 sub-inputs, therefore sees a larger reduction in latency. For the Taobao Alibaba dataset, each input consists of a stream of up to 21 sub-inputs, therefore sees a larger reduction in latency [5].

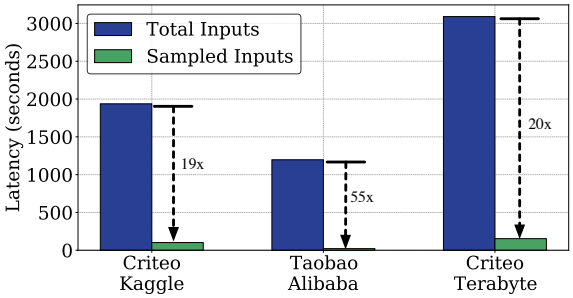


Figure 5.2: Reduction in the profiling latency when input dataset is sampled for embedding table access pattern.

5.1.3 Categorize and determine hot embedding size with the Profiler

We perform profiling before the training begins. The goal of the *profiler* is twofold - (1) for the sampled input dataset $\widehat{\boldsymbol{D}}$ it creates an access profile of each embedding table ($\boldsymbol{E}_z$), where $\boldsymbol{z}$ is the table number and (2) it further samples this access profile to determine what the size of the hot embedding table.

Embedding Logger: The ***profiler*** uses an embedding logger for each table to keep track of access counts (denoted as $\boldsymbol{k}$) of $\widehat{\boldsymbol{D}}$ into each entry in $\boldsymbol{E}_z$. As each model can access multiple embedding tables, our implementation assumes any table that is greater than or equal to 1 *MB* to be large. Embedding tables smaller than 1MB are de facto considered "hot" as they can easily fit even on low-end GPUs. Thereafter, the *profiler* would still need to estimate the hot embedding table sizes, *without* traversing all the embeddings.

Estimating the hot embedding table sizes per threshold: *Profiler* creates a sampled access profile for each embedding table entry across all the tables by selecting random chunks of embedding entries and their observed access pattern from the logger. This enables estimating the size of the hot embeddings without traversing all the tables in their entirety. As the embedding logger observes only $\boldsymbol{x}\%$ of the actual inputs, we need to *scale down* the required access counts to classify hot data. For embedding table number $\boldsymbol{z}$ and a threshold $\boldsymbol{t}$, the new hot embedding cutoff for each *sampled entry* is denoted by $\boldsymbol{H}_{zt}$, described in Equation 5.1:

$$H_{zt} = t \times T \times \frac{x}{100} \tag{5.1}$$

We then pick $\boldsymbol{n}$ random samples, each consisting of $\boldsymbol{m}$ = 1024 entries entries from embedding logger for table z. Our implementation uses $\boldsymbol{n}$ = 35 and each sample consists of $\boldsymbol{m}$ = 1024 embedding entries.

This chunk based sampling allows us to create a distribution of the access pattern. Our work uses Central Limit Theorem (CLT) to estimate the mean of the parent distribution. CLT has the property that, *irrespective of the parent distribution*, the mean of the sampled distribution will always approach the mean of the parent distribution. This is because, when the sample size $\boldsymbol{n} \geq 30$, CLT considers the sample size to be large and the sampled mean will be normal even if the sample does not originate from a Normal Distribution [39]. As each embedding sample

chunk consists $\boldsymbol{m}$ = 1024 entries, we can estimate the actual embedding table size with a precision of $\frac{1}{1024}$. For each chunk, we count ($\boldsymbol{C}$) the number of entries with access counts ($\boldsymbol{k}$) greater than or equal to $\boldsymbol{H_{zt}}$. This is represented by Equation 5.2:

$$C_i = \sum_{j=1}^{m} (k_j \geqslant \mathrm{H}_{zt}) \tag{5.2}$$

For $\boldsymbol{n}$ chunks, the standard deviation is $\boldsymbol{s}$ and the mean is $\bar{\boldsymbol{y}}$, shown by Equation 5.3:

$$\bar{y} = \frac{\sum_{i=1}^{n} C_i}{n} \tag{5.3}$$

Figure 5.3 shows the latency savings from sampling embedding table instead of iterating through all the embedding access content.

As the *profiler* scans 14x fewer embedding entries for each t it reduces latency of each scan by 14.5×-61×.

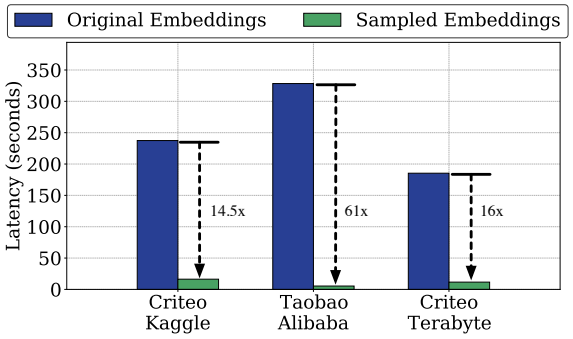


Figure 5.3: Reduction in the *latency per iteration* by using to estimate the hot embedding size per threshold. The *total* latency to scan all embedding tables is under 25 seconds per threshold iteration.

5.2 Confidence in the estimated embedding table size.

The goal of the *profiler* is to establish confidence in the estimated embedding size. A confidence interval, in statistics, refers to the probability $(1-\alpha)$ that a population parameter will fall between a set of values. To compute the confidence interval for the *profiler*'s estimated embedding table size, FAE uses the standard 'Student's t-

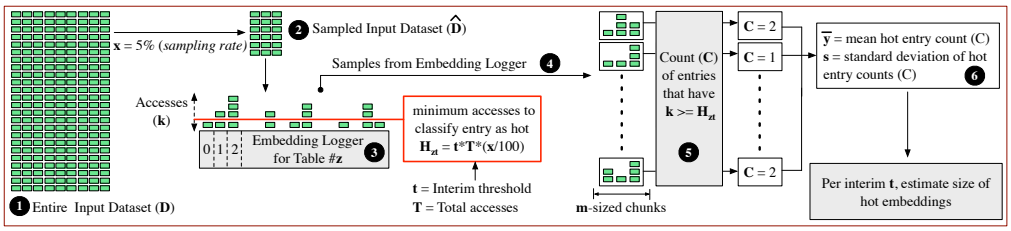


Figure 5.4: The flow of events in Input Sampler and Profiler. The original input 1 is sampled 2 at 5%. This sample is used by the *profiler* to create an access profile across embedding entries in the logger 3. For each threshold, A few chunks from the embedding logger is randomly sampled 4 to estimate the count of hot entries 5. The mean and standard deviation of this count determines the size of hot embedding tables per threshold 6.

interval'. As $\bar{y}$ follows a t-distribution, the 100×(1-α) confidence interval (CI) for $\bar{y}$ is represented by Equation 5.4:

$$CI_{100\times(1-\alpha)} = \bar{y} \pm t_{\frac{\alpha}{2}} \times \sqrt{(\frac{N-n}{N}) \times (\frac{s^2}{n})} \tag{5.4}$$

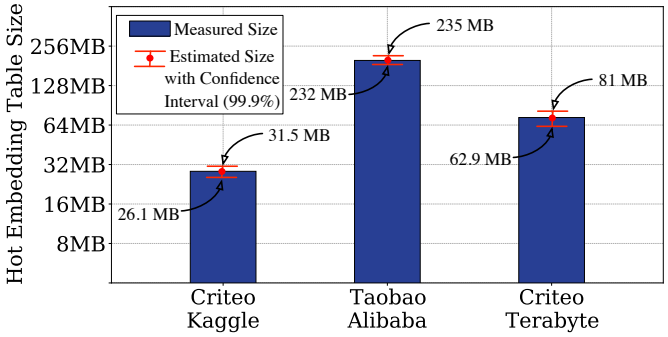


Figure 5.5: Estimated sizes of hot embedding tables with . For a confidence interval of 99.9%, the estimation is within 10% (upper bound) of the actual size.

Figure 5.5 shows the estimation variability compared to the actual values for a confidence interval of 99.9%. Actual value of the hot embedding size is exact size the *profiler* would have obtained if it had processed the entire access pattern for each embedding table. This variability can be reduced if we specify a smaller confidence interval. We observe that the estimated values are within 10% of the

actual values. As such, for every threshold, the *profiler* process described above is executed to determine the size of the hot embeddings.

The Statistical Optimizer, based on this size and user requirements, either accepts the threshold or tunes it further as described below. Our experiments show that allocated memory of $\boldsymbol{L}$ = 512MB suffices for most GPUs (including low-end GPUs).

Converging on a Threshold using Statistical Optimizer

The Statistical Optimizer invokes the profiler with varying $\boldsymbol{t}$ (interim thresholds) and a desired confidence interval to determine the final $\boldsymbol{t}$. Based on the embedding size estimated for an interim threshold, the optimizer tunes the threshold to be higher or lower than the previous.

This ensures that the threshold is tuned appropriately based on the available GPU memory for each model architecture. The Statistical Optimizer then provides the final threshold as output to the next blocks in the FAE.

5.3 Input and Embedding Classifier

The *embedding classifier* uses the output of the Embedding Logger and the final threshold from Statistical optimizer to tag (hot or cold) the embedding table entries. This requires *only* one pass of each embedding table. Additionally, the *input classifier* uses the final access threshold value and accesses to the already classified embedding table to identify hot sparse-feature inputs. Typically, there are 10s of embedding tables in a recommender model. A sparse-feature input typically accesses one or more entries in each of these embedding tables. A sparse-feature input is classified as hot only if all its embedding table accesses are to hot entries.

This component typically requires only *one* pass of the entire sparse-feature input (S_I) and just checks if the embedding entry indices are present in the hot-embedding bags. Hot-embedding bag is composed of all index of all hot embedding table entries across all embedding tables in the model. As this is completely parallelizable operation across both inputs and embedding indices, we divide this task across multiple cores in the CPU. For a 16 core machine (32 hardware threads), the total time for this phase for different access thresholds is given by Figure 5.6.

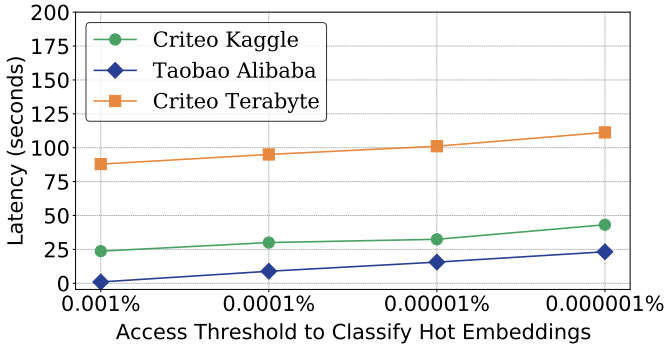


Figure 5.6: The latency of the input processor from dataset to classify sparse-feature inputs (as hot or cold) as we vary the access threshold. Overall, even for very low access thresholds, we only require only a maximum of 110 seconds.

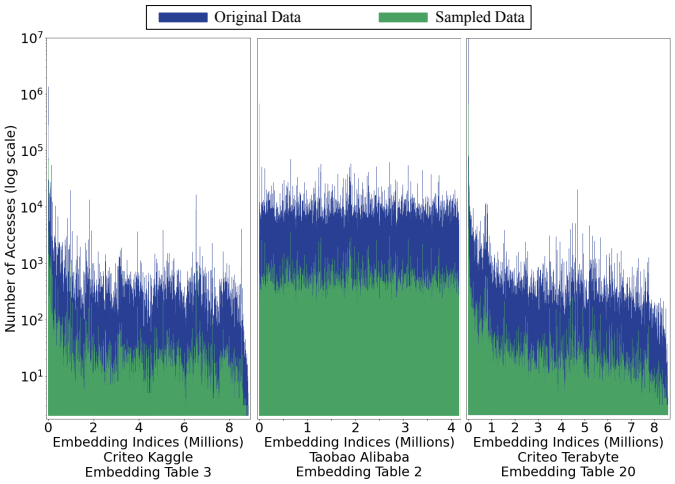


Figure 5.7: Embedding table access profile from the original inputs ($\boldsymbol{D}$) and the sampled inputs ($\widehat{\boldsymbol{D}}$) – sampling rate ($\boldsymbol{x}$) = 5%. We observe that $\widehat{\boldsymbol{D}}$ has a similar access signature to $\boldsymbol{D}$.

The *input classifier* also bundles hot and cold inputs together into minibatches. As aforementioned, we require the entire minibatch to be hot to avoid the data shuffling between CPU and GPU. If an minibatch of inputs is entirely hot, the entire execution can happen in a data-parallel mode on the GPU without any interference from the CPU. Once we have preprocessed the sparse-input data into hot and cold minibatches, we store this in the FAE format for any subsequent training runs.

5.4 Scheduler for Dynamic Hot-Cold Swaps

FAE's preprocessing provides a dataset that is distributed into hot and cold mini-batches and a set of hot embeddings. The embedding replicator replicates the hot embedding across all GPUs. In addition to that, hot embeddings also are available on CPU for baseline model execution using cold input. Next, we discuss the runtime scheduling of hot and cold mini-batches to ensure the baseline accuracy metrics whilst providing accelerated performance.

In the most basic form, FAE can schedule the entire collection of minibatches comprising hot inputs followed by cold inputs, or vice versa, but such a schedule can have potential impact on training accuracy. This is because the hot inputs only access and update the hot embedding entries, and training using only hot inputs for a long time can potentially reduce the randomness in training.

For non-convexity loss optimization problems, this makes gradient descent based algorithms susceptible to local minima. To mitigate this, machine learning community has often deployed data shuffling. Next, we discuss how we uniquely attenuate this issue for our framework.

5.5 Communication Overheads

To re-introduce randomness in our training while also attaining accelerated performance, we intermittently schedule hot and cold minibatches. However, changing input type (hot vs cold) can degrade performance as each of these events requires synchronization of hot embedding parameters between CPU and GPU copies. To balance this trade off, we offer Shuffle Scheduler that dynamically determines the interleaving of hot and cold mini-batches based on the runtime training metric. The scheduler always begins with training on cold inputs as they update a wider range of embedding entries, albeit infrequently per entry. The rate of scheduling hot and cold minibatches can be tuned dynamically based on Equation 5.5. In the equation, $r(i)$ is the rate at i^{th} swap. Rate of $(R(100))$ implies that 100% of the mini-batches of cold inputs will be completed before the first hot mini-batches is issued. A rate of $(R(1))$ implies hot and cold are shuffled after every mini-batch. $Test_L$ is the testing loss and u is a count of swaps.

$$r(i+1) = \begin{cases} min(r(i) * 1/2, R(1)) & if \Delta Test_L(i) \geq Test_L(i-1) \\ max(r(i) * 2, R(100)) & if \Delta Test_L(i) \leq Test_L(i-u) \\ r(i) & otherwise \end{cases} \tag{5.5}$$

Based of the testing loss obtained after each swap, we determine whether the rate needs to be changed based on the following two conditions.

Testing loss used for the scheduler can use loss functions such as mean squared loss and cross-entropy logarithmic loss, based on the model requirement. All of our models and their datasets use the logarithmic loss to establish the efficacy of training. Nonetheless, as we perform a comparison of loss score between each subsequent swap, any loss function can be used. If the framework observes an increase in the test loss from the previous schedule, it immediately reduces the rate by half. This implies that the remaining minibatches of hot and cold inputs will be split into an alternate of cold and hot schedules. The rate can be reduced to a minimum of $R(1)$. If the test loss decreases, rate remains unchanged, as this is the expected behaviour, unless the loss has been decreasing successively for u schedules. This is the second case where rate is changed, i.e., increased by 2, up to a max of $R(100)$. Similar to prior work that offers automatic convergence checks to avoid over-fitting, the downward trend of test loss curve [49] consecutively for 4 strips shows a balance between redundancy, badness, and slowness; thus we choose u as 4. Apart from the above two cases, the rate remains unchanged. The Shuffle Scheduler ensures that accuracy remains the priority of FAE. FAE begins training with $R(50)$ (alternate cold and hot mini-batches) for a dataset, and tunes the rate accordingly.

5.6 Summary

In this chapter, we present the flow of Frequently Accessed Embeddings. FAE comprises of classifier with a threshold to segregate the hot and cold entries. Also, FAE benefits from scheduler to dynamically swap the hot and cold. Another major component is a shuffler that dynamically determines the execution order of hot and cold sparse input mini-batches across the CPU and GPUs at runtime.

Chapter 6

Evaluation

In Chapter 5, we introduced FAE to improve the end-to-end training time for Deep Learning Recommendation Models. In this chapter, we showcase the efficacy of our FAE framework on 4 real world datasets, using established recommendation models RMC1, RMC2, RMC3, and RMC4 which represent four classes of at-scale models [19]. We prototype FAE on top of the widely used open source implementation of Deep Learning Recommendation System (DLRM) [41] [1] and Time-based Sequence Model for Personalization and Recommendation Systems(TBSM) [26] [2] frameworks to train recommender models.

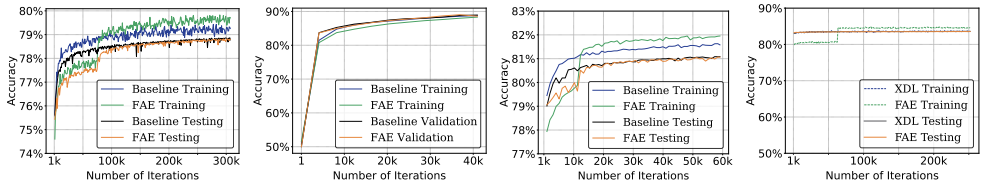
Based on the sparse input configuration, there is a model-dataset correspondence, RMC1 model on Taobao Alibaba [5] with TBSM, and RMC2 on Criteo Kaggle [12], RMC3 on Criteo Terabyte [13], and RMC4 on Avazu [32] set that provides temporal user behavior data, it is the only dataset that can leverage the TSL layer in TBSM.

Table 6.1 describes the details of the model architecture for RMC1, RMC2, RMC3, and RMC4 including their dense and sparse features, embedding table numbers and size, and neural network configurations. In addition to these real world datasets and their corresponding models, we also perform evaluation on synthetic models to show case the efficacy of our framework.

[1] https://github.com/facebookresearch/dlrm
[2] https://github.com/facebookresearch/tbsm

Table 6.1: Model Architecture Parameters and Characteristics of the Datasets for our Workloads

Workload	Dataset	Training Input		Dataset Features		Embedding Tables			Neural Network Config		
		Number	Size	Dense	Sparse	Rows	Rows Dim	Size	Bottom MLP	Top MLP	DNN
RMC1 (TBSM [26])	Taobao (Alibaba) [5]	10 M	1 GB	3	3	5.1M	16	0.3 GB	1-16 & 22-15-15	30-60-1	Attn. Layer
RMC2 (DLRM [41])	Criteo Kaggle [12]	45 M	2.5 GB	13	26	33.8M	16	2 GB	13-512-256-64-16	512-256-1	-
RMC3 (DLRM [41])	Criteo Terabyte [13]	80 M	45 GB	13	26	266M	64	63 GB	13-512-256-64	512-512-256-1	-
RMC4 (DLRM [41])	Avazu [32]	32.3 M	2.4 GB	1	21	9.3M	16	0.55 GB	1-512-256-64-16	512-256-1	-



(a) Criteo Kaggle **(b)** Taobao Alibaba **(c)** Criteo Terabyte **(d)** Avazu

Figure 6.1: Increasing Accuracy with training iterations when optimized with FAE framework. As we see, all the datasets and corresponding recommender models achieve the XDL accuracy for both training and test or validation sets.

6.1 Experimental Setup

6.1.1 Software libraries and setup

We showcase the efficacy of our framework on 4 real world datasets, using established recommendation models RMC1, RMC2, RMC3, and RMC4 which represent four classes of at-scale models [19]. We prototype on top of the widely used open source implementation of Deep Learning Recommendation System [41] [3] and Time-based Sequence Model for Personalization and Recommendation Systems (TBSM) [26] [4] frameworks to train recommender models.

Based on the sparse input configuration, there is a model-dataset correspondence, RMC1 model on Taobao Alibaba [5] with TBSM, and RMC2 on Criteo

[3] https://github.com/facebookresearch/dlrm
[4] https://github.com/facebookresearch/tbsm

Kaggle [12], RMC3 on Criteo Terabyte [13], and RMC4 on Avazu [32] all with DLRM. TBSM consists of embedding layer and time series layer (TSL); the embedding layer is implemented through DLRM. TSL resembles an attention mechanism and contains its own MLP network to compute one or more context vectors between history of items and the last item. As Taobao Alibaba is the only dataset that provides temporal user behavior data, it is the only dataset that can leverage the TSL layer in TBSM.

Table 6.1 describes the details of the model architecture for RMC1, RMC2, RMC3, and RMC4 including their dense and sparse features, embedding table numbers and size, and neural network configurations.

In addition to these real world datasets and their corresponding models, we also perform evaluation on synthetic models to show case the efficacy of our framework. As relies on popularity of certain inputs, we execute these synthetic models on Criteo Terabyte (largest dataset) to ensure the semantics of the training input.

6.1.2 Server Architecture

The base DLRM and TBSM code is configured using the Pytorch-1.7 and executed using Python-3. We use the torch.distributed back end to support scalable distributed training and performance optimization [48]. NCCL is used [43] for gather, scatter, and all-reduce collective calls via the backend NVLink [2] interconnect. DLRM and TBSM are also implemented on XDL 1.0 using Tensorflow-1.2 as the computation backend.

Table 6.2: System Specifications

Device	Architecture	Memory	Storage
CPU	Intel Xeon Silver 4116 (2.1GHz)	768 GB DDR4 (2.7GB/s)	1.9 TB NVMe SSD
GPU	Nvidia Tesla V100 (1.2GHz)	16 GB HBM-2.0 (900GB/s)	-

Table 6.2 describes the configuration of our datacenter servers. These servers comprise 24-core Intel Xeon Silver 4116 (2.1 GHz) processor with Skylake architecture. Each server has a DRAM memory capacity of 192 GB. Each DDR4-2666 channel has 8 GB memory. Each server also has a local storage of 1.9 TB NVMe SSD. Each server offers 4 NVIDIA Tesla-V100 each with 16GB memory capacity

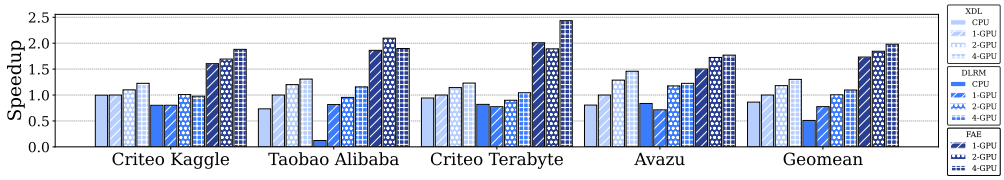


Figure 6.2: The performance of Criteo Kaggle, Taobao Alibaba, Criteo Terabyte, and Avazu training with the vs XDL and DLRM. All values are normalized to XDL 1-GPU.

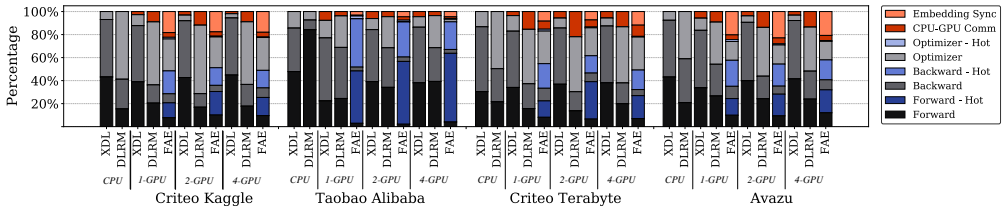


Figure 6.3: Latency breakdown for the 1, 2, and 4 GPU executions. The framework adds the overhead of embedding synchronization across CPUs and GPUs, not present in XDL and DLRM.

as a general purpose GPU. The GPUs are connected using the high speed NVLink-2.0 interconnect. Every GPU is communicating with the rest of the system via a 16x PCIe Gen3 bus. In this paper, we perform experiments on a single server with a maximum of 4 GPUs. We expect our insights to hold true even in a multi-server scenario.

6.1.3 Baselines and terminology

We compare FAE optimized training against two baselines: (1) open sourced implementation of DLRM and TBSM and (2) DLRM and TBSM implementation on XDL [30]. For both the baselines we execute on CPU only mode and CPU-GPU hybrid mode with varying number of GPUs. The CPU only mode is referred to as XDL-CPU and DLRM-CPU. For CPU-GPU hybrid mode, in case of DLRM, embeddings execute on CPU.

6.2 Accuracy

In this section, we evaluate the accuracy of FAE and the end-to-end training time and comparing it against the two baselines.

Table 6.3: Accuracy Metric Comparisons

2***Dataset**	**XDL**			**FAE**		
	Accuracy (%)	**AUC**	**Logloss**	**Accuracy (%)**	**AUC**	**Logloss**
Criteo Kaggle	78.86	0.802	0.452	78.86	0.802	0.452
Taobao Alibaba	89.21	-	0.269	89.03	-	0.271
Criteo Terabyte	81.07	0.802	0.424	81.06	0.802	0.424
Avazu	83.61	0.758	0.390	83.60	0.758	0.391

6.2.1 Accuracy Results

Figure 6.1 illustrates the accuracy of Criteo Kaggle, Taobao Alibaba, Criteo Terabyte, and Avazu for their RMC2,RMC1, RMC3, and RMC4 models. We use a full-precision XDL-CPU execution baseline that exactly follows the DLRM and TBSM executions. 6.3 compares the accuracy metrics for all the work-loads.

We use testing accuracy, Area Under Curve (AUC), and cross-entropy loss (log loss) as recommendation model performance metric, as established by the MLPerf [1] community. For Taobao dataset, we use the accuracy and log loss as performance metric,as AUC is not offered.

FAE training reduces the average execution time (geomean) by 42%, 36%, and 34%, 1-GPU, 2-GPU, and 4-GPU executions, respectively. The GPU comparisons assume same number of GPUs for XDL and FAE.

FAE observes an initial jump in accuracy for both Criteo and Avazu datasets after the first swap between cold and hot mini-batch. Once, model is trained on both the types of mini-batches, we do not observe any more jumps. As we interleave it with the first hot mini-batch, many of pertinent embedding entries get updated and we reach the baseline accuracy for both training and testing sets.

As the table shows, each model achieves the corresponding baseline accuracy. For all the datasets, we observe that when the Shuffle Scheduler alternately issues cold and hot minibatches at $R(50)$, the models are able to converge to the baseline accuracy in the same number of baseline training iterations.

FAE observes an initial jump in accuracy for both Criteo and Avazu datasets after the first swap between cold and hot mini-batch. Once, model is trained on both the types of minibatches, we do not observe any more jumps. As we interleave it with the first hot mini-batch, many of pertinent embedding entries get updated and we reach the baseline accuracy for both training and testing sets.

Performance Gains and Absolute Training Times

Figure 6.2 shows the performance improvement of end-to-end training execution using FAE in comparison to XDL and DLRM/TBSM. The end-to-end training runs are terminated when the established accuracy metric (cross-entropy loss or area under the curve) is met. The performance is normalized to XDL 1-GPU execution For a singled device (CPU or 1-GPU), we use a mini-batch of 1K, 256, 1K and1K inputs for Criteo Kaggle, Taobao Alibaba, Criteo Terabyte and Avazu, respectively. FAE training reduces the average execution time (geomean) by 42%, 36%, and 34%, 1-GPU, 2-GPU, and 4-GPU executions, respectively. The GPU comparisons assume same number of GPUs for XDL and FAE.

We maintain weak scaling across distributed runs where the minibatch size is scaled with the number of GPUs. For example, 2 GPU execution use 2K, 512, 2K and 2K mini-batch size for Criteo Kaggle, Taobao Alibaba, Criteo Terabyte and Avazu, respectively.

For Taobao, 4 GPU execution takes more time than 2 GPU execution because the dataset is relatively small, thus the cold minibatch executions overshadow benefits of FAE. Overall FAE reduces the training time by 2.3× and 1.52× in comparison to XDL CPU-only and XDL CPU-GPU with 4-GPUs.

Table 6.4: Absolute Training Time for 10 Epochs (mins)

Dataset	**XDL**	**1-GPU**		**2-GPU**		**4-GPU**	
	CPU	**XDL**	**FAE**	**XDL**	**FAE**	**XDL**	**FAE**
Criteo Kaggle	197.56	196.97	122.71	179.16	116.27	160.65	104.69
Taobao Alibaba	1108.84	813.10	436.58	677.00	387.79	621.96	428.55
Criteo Terabyte	404.25	380.88	189.73	330.06	201.61	309.51	156.45
Avazu	134.28	108.24	72.07	84.04	62.73	74.20	61.15

Absolute time: We compare the absolute end-to-end training time, when all the executions reach their required accuracy metric. These times are shown in minutes in 6.4. We use minibatch of 1k, 2k,and 4k for Crtieo Kaggle, Terabyte, and Avazu and 256, 512 and1k for Taobao Alibaba dataset. We observe that the RMC1 model with Taobao obtains most benefits from general GPU acceleration as it employs a relatively large deep learning neural network.

FAE can further accelerate the training of this model and reduce the training time to 428 minutes with 4-GPU FAE compared to 621 minutes with 4-GPU XDL. With the recent developments in machine learning [58], we expect the neural networks to increase inconsiderably in size for recommender models. Results clearly show that FAE can enable GPU acceleration without incurring large data transfer overhead between CPU and GPU.

Latency breakdown

Figure 6.3 shows the breakdown of the total runtime for each of the workloads executing on CPU-only and 1, 2, and 4 GPUs. In the Figure, colors for cold inputs are consistent across XDL, DLRM, and FAE executions. As the figure shows,the optimizer constitutes a large portion of the DLRM execution. This is because, a CPU cannot efficiently execute the massively parallel optimizer operation.

FAE is able to mitigate some of these inefficiencies and reduce the optimizer time by performing both the neural network and embedding updates on GPUs for the hot input min-batches. In case of XDL, efficiency of Advanced Model Server (AMS) is improved using GPU to speed up the massively parallel optimizer and embedding dictionary lookup. Even XDL is limited by the size of GPU memory, hence only the index of embedding dictionary is stored in GPU memory.

Due to small size of hot embedding tables, FAE stores the entire table in GPU memory instead of only the indices. Hence, for FAE the optimizer time for hot mini-batches is significantly lower than the cold mini-batches, as the of hot inputs are observed more often but accelerated on GPU.

6.3 also shows the percentage of time spent by XDL,DLRM/TBSM, and FAE on data transfer to and from the embedding layers. This data transfer is completely eliminated for FAE for hot mini-batches. For DLRM/TBSM implementations, the

data transfer time comprises the time spent on transferring embedding data to the GPU. For XDL, the time reported is spent on transferring embedding indices and model dense parameters to the GPU.

Embedding Synchronization

One overhead imposed by FAE is one form of embedding synchronization while switching between cold and hot minibatches. The embedding tables are updated across CPU and GPU memories to ensure the training process observes the same entries. This overhead is shown by the embedding sync on FAE executions and is highlighted in the Figure 6.3.

Avazu observes a higher percentage of embedding synchronization overhead because of its comparatively smaller embedding size. Thus the fixed transfer cost from CPU to GPU, using PCIe, is not amortized over a large data transfer. On the contrary, Taobao observes the least percentage of synchronization overhead. This can be attributed to the high percent of forward and backward time of the Taobao RMC1 recommender model due to its deep attention layer. Thus, as the recommender models become bigger with larger embedding tables and deeper neural network layers, FAE can offer higher benefits by reducing the CPU-GPU data transfer between embedding and DNN layers, whilst observing amortized embedding synchronization overheads. This is because, even though embedding tables are expected to increase in size, a larger absolute size of embedding a does not necessarily imply a proportionally large hot embedding table. This is because certain inputs are always going to be way more popular than the others.

Table 6.5: CPU-GPU data transfer time for 10 Epochs (mins)

Dataset	**1-GPU**			**2-GPU**			**4-GPU**		
	DLRM	**XDL**	**FAE**	**DLRM**	**XDL**	**FAE**	**DLRM**	**XDL**	**FAE**
Criteo Kaggle	22.09	5.39	4.99	23.12	5.61	4.35	18.00	3.05	4.29
Taobao Alibaba	37.93	24.97	3.24	38.27	12.89	11.11	25.04	6.24	6.04
Criteo Terabyte	76.01	13.46	13.27	92.98	18.94	12.41	48.43	17.49	15.24
Avazu	13.94	6.23	2.97	12.68	3.19	3.17	11.94	2.36	2.79

Table 6.6: Amount of Data Transferred over 10 Epochs

Dataset	DLRM (GB)	XDL (GB)	FAE (GB)
Criteo Kaggle	60.89	23.16	14.99
Taobao Alibaba	1.95	0.51	0.61
Criteo Terabyte	375.06	95.60	69.58
Avazu	40.45	30.27	10.45

6.2.2 Data transfer between CPU and GPU

Table 6.5 shows the absolute communication time to transfer the embedding layers and Table 6.6 shows the amount of data transferred for XDL, DLRM/TBSM and FAE execution including the embedding synchronization for FAE. On average FAE reduces the total data transfer from 37 GB with XDL to 24 GB, even including the embedding synchronization overhead. Which translates to 12% improvement in CPU-GPU data transfer time. In case of XDL, all dense parameters needs to be transferred from AMS to backend workers and vice versa pertraining iteration. FAE only require parameters to be transferred between CPU and GPU across the hot and cold minibatch swap.

Performance improvement with varying mini-batch size

6.4 shows the performance benefits of FAE training over XDL execution for a 4-GPU system. Speedup is normalized to XDL execution with mini-batch size of 1K, 256, 1K and 1K for Criteo Kaggle,Taobao Alibaba, Criteo Terabyte and Avazu datasets respectively. As the mini-batch size increases, we observe higher benefits because the overheads of FAE are amortized over a larger input set. For instance, now the Embedding Replicator replicates the model fewer times. However, with XDL, we do not see such an improvement because of extra time being spent on creating and sending larger minibatches to the backend workers.

Performance improvement for synthetic models

We use real-world training data as FAE utilizes the property that certain inputs are way more popular than the others. However, to understand the efficacy of FAE on varying types of model architectures, we create synthetic configurations, shown in

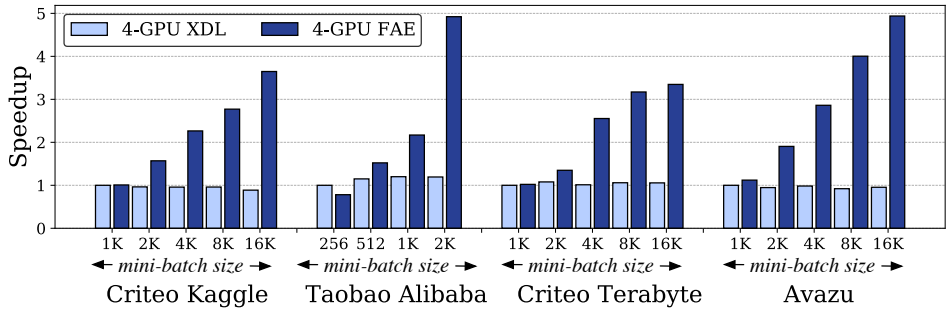


Figure 6.4: Speedup of with varying mini-batch sizes for a 4-GPU system, compared to a 4-GPU XDL.

Table 6.7, that can be executed on Criteo Terabyte dataset. Figure 6.5 shows the performance improvements of FAE across various synthetic models over XDL and DLRM/TBSM. FAE provides on average 2.94× speedup across small and large synthetic models as compared to XDL.

Table 6.7: Synthetic Models' Configuration

Dataset	Bottom MLP	Top MLP
SYN-M1	13-64	512-1
SYN-M2	13-512-64	512-256-1
SYN-M3	13-1024-512-64	512-1024-256-1
SYN-M4	13-1024-512-256-64	512-1024-512-256-1

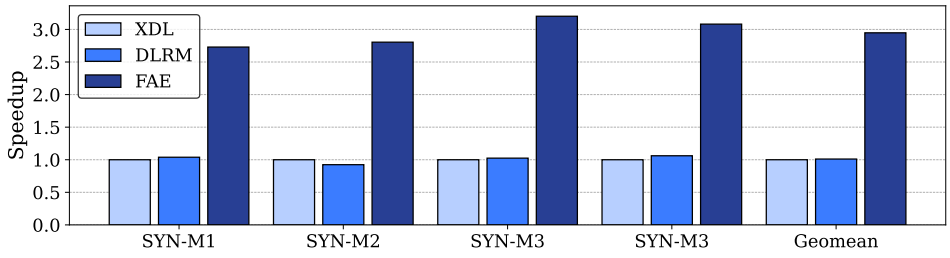


Figure 6.5: Performance comparison of with XDL 4-GPU across various synthetic models.

Power Benefits

Table 6.8 shows the per GPU power consumption using the baseline and FAE for a 1024 mini-batch. FAE reduces GPU power consumption by 9.7% in comparison

to XDL. This is primarily due to the reduced communication cost between devices.

Table 6.8: GPU Power Consumption Comparison

Dataset	XDL	DLRM	FAE
Criteo Kaggle	61.83W	58.91W	55.81W
Alibaba	56.39W	60.21W	56.62W
Criteo Terabyte	59.71W	62.47W	57.03W
Avazu	60.2W	58.03W	56.4W

6.3 Summary

In this chapter, we evaluate the accuracy and performance of the proposed framework FAE. In addition, We compare FAE optimized training against two baselines, open sourced implementation of DLRM and TBSM and (2) DLRM and TBSM implementationon XDL. For both the baselines we execute on CPU only mode and CPU-GPU hybrid mode with varying number of GPUs. We do the evaluation in terms of accuracy and absolute timing as well the latency breakdown. FAE achieves the average speedup across all 2.34x speedup while maintaining the same accuracy. Overall FAE reduces the training time by 2.3× and 1.52× in comparison to XDL CPU-only and XDL CPU-GPU with 4-GPUs.

Chapter 7

Conclusion and Future Work

7.1 Summary

Recommendation models aim to learn user preferences and provide a targeted user experience by employing very large embedding tables. Even though these tables often cannot fit on GPU memory,these models also comprise neural network layers that are well suited for GPUs. These contrasting requirements splits the training execution on CPUs (for memory capacity) and GPUs (for compute throughput). Fortunately, for real-world data, we observe that embedding tables exhibit a skewed data access pattern. This can be attributed to certain training inputs (users and items) that are much more popular than the others.

This observation allows us to develop a comprehensive framework, namely FAE, that uses statistical techniques to quantify the hotness of embedding entries based on the input dataset. This hotness of embedding tables in turn allows the framework to optimally layout embedding so that the GPU memory is efficiently utilized to store highly accessed data close to the compute. To capture most of the performance benefits, FAE bundle shot inputs and cold inputs in separate mini-batches. This helps FAE accelerate the hot mini-batch by executing the whole model on GPU and eliminate any CPU-GPU embedding data transfers.

The training for these hot inputs happens entirely on GPUs, thus reducing any CPU-GPU communication overhead between CPU-GPU and GPU-GPU from embedding and neural network layers. Our experiments on DLRM and TBSM recom-

mender models with real datasets show that FAE reduces the overall training time by 2.3× and 1.52×in comparison to XDL CPU-only and XDL CPU-GPU execution while maintaining baseline accuracy.

7.2 Future Work

Future work relies on investigating first new models to this aim second different representations for the same purpose like graph processing. Using graph representation, a whole different class of problems arises. For the same model future works effort falls into categories such as finding other patterns other than popularity embodies in the recommendation dataset or model. Other orthogonal areas of research can be investigating new methods of compression to reduce the size of DRLM further or deploying some federated learning techniques to overcome the existing issues.

Bibliography

[1] Mlperf becnhmarks. https://mlcommons.org/en/training-normal-10/. → page 44

[2] Nvlink. URL https://developer.nvidia.com/nccl. → pages 28, 42

[3] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale, 2020. → page 21

[4] A. O. D. V. Alexander Novikov, Dmitry Podoprikhin. Tensorizing neural networks, 2015. URL https://arxiv.org/abs/1509.06569. → page 11

[5] Alibaba. User behavior data from taobao for recommendation. https://tianchi.aliyun.com/dataset/dataDetail?dataId=649&userId=1. → pages 32, 40, 41

[6] J. L. W. L. K.-C. L. J. X. B. Z. Bencheng Yan, Pengjie Wang. Binary code based hash embedding for web-scale applications, 2021. URL https://arxiv.org/abs/2109.02471. → page 14

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014. → pages 1, 9

[8] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016. → page 1

[9] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX

Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi. → page 23

[10] E. Chung, J. Fowers, K. Ovtcharov, , A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, G. Weisz, M. Haselman, and D. Zhang. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38: 8–20, March 2018. URL https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/. → page 1

[11] X. L. C.-J. W. Chunxing Yin, Bilge Acun. Tt-rec: Tensor train compression for deep learning recommendation models, 2021. URL https://arxiv.org/abs/2101.11714. → pages 4, 12

[12] CriteoLabs. Criteo display ad challenge, . https://www.kaggle.com/c/criteo-display-ad-challenge. → pages 40, 41, 42

[13] CriteoLabs. Terabyte click logs, . https://labs.criteo.com/2013/12/download-terabyte-click-logs. → pages 40, 41, 42

[14] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. Hazelwood, A. Cidon, and S. Katti. Bandana: Using non-volatile memory for storing deep learning models. *Proceedings of Machine Learning and Systems*, 1:40–52, 2019. → page 16

[15] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *International Symposium on Field-Programmable Custom Computing Machines*. IEEE, May 2014. URL http://research.microsoft.com/apps/pubs/default.aspx?id=217166. → page 22

[16] H. N. Fumito Yamaguchi. Hardware-based hash functions for network applications. 2013 19th IEEE International Conference on Networks (ICON), Singapore, 2013. IEEE. doi:10.1109/ICON.2013.6781990. URL https://ieeexplore.ieee.org/document/6781990. → page 14

[17] A. Ginart, M. Naumov, D. Mudigere, J. Yang, and J. Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. *ArXiv*, abs/1909.11810, 2019. → page 17

[18] C. A. Gomez-Uribe and N. Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), Dec. 2016. ISSN 2158-656X. doi:10.1145/2843948. URL https://doi.org/10.1145/2843948. → page 1

[19] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501, 2020. doi:10.1109/HPCA47549.2020.00047. → pages 4, 40, 41

[20] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. → page 22

[21] C. W. X. L. J. T. Haochen Liu, Xiangyu Zhao. Automated embedding size search in deep recommender systems. SIGIR '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3397271.3401436. URL https://doi.org/10.1145/3397271.3401436. → pages 12, 14

[22] C. W. X. L. J. T. Haochen Liu, Xiangyu Zhao. Automated embedding size search in deep recommender systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, CIKM '20, page Pages 2307–2316, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3397271.3401436. URL https://doi.org/10.1145/3397271.3401436. → pages 11, 12, 14

[23] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi:10.1145/3038912.3052569. URL https://doi.org/10.1145/3038912.3052569. → pages xi, 3

[24] J. Huang, J. Park, P. T. P. Tang, A. Tulloch, et al. Mixed-precision embedding using a cache. *arXiv preprint arXiv:2010.11305*, 2020. → page 2

[25] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. Gpipe: Efficient training of giant

neural networks using pipeline parallelism. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 103–112, 2019. URL http://papers.nips.cc/paper/8305-gpipe-efficient-training-of-giant-neural-networks-using-pipeline-parallelism. → pages 9, 23

[26] T. Ishkhanov, M. Naumov, X. Chen, Y. Zhu, Y. Zhong, A. G. Azzolini, C. Sun, F. Jiang, A. Malevich, and L. Xiong. Time-based sequence model for personalization and recommendation systems. *CoRR*, abs/2008.11922, 2020. URL https://arxiv.org/abs/2008.11922. → pages xi, 1, 3, 4, 40, 41

[27] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2018. doi:10.1109/ISCA.2018.00070. → page 22

[28] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 947–960, USA, 2019. USENIX Association. ISBN 9781939133038. → page 9

[29] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019. → page 24

[30] B. Jiang, C. Deng, H. Yi, Z. Hu, G. Zhou, Y. Zheng, S. Huang, X. Guo, D. Wang, Y. Song, L. Zhao, Z. Wang, P. Sun, Y. Zhang, D. Zhang, J. Li, J. Xu, X. Zhu, and K. Gai. Xdl: An industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, DLP-KDD '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367837. doi:10.1145/3326937.3341255. URL https://doi.org/10.1145/3326937.3341255. → pages iii, 4, 21, 43

[31] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski,

A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928. doi:10.1145/3079856.3080246. URL https://doi.org/10.1145/3079856.3080246. → page 1

[32] Kaggle. Avazu mobile ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction. → pages 40, 41, 42

[33] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. Wu, M. Hempstead, and X. Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020. doi:10.1109/ISCA45697.2020.00070. → page 17

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi:10.1145/3065386. URL https://doi.org/10.1145/3065386. → page 23

[35] A. V. Kumar and M. Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/kumar. → page 10

[36] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proc. VLDB Endow.*, 9(14):1647–1658, Oct. 2016. ISSN 2150-8097. doi:10.14778/3007328.3007331. URL https://doi.org/10.14778/3007328.3007331. → pages 19, 23

[37] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. Mar. 2016. → page 1

[38] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. Analyzing and mitigating data stalls in dnn training. In *VLDB 2021*, January 2021. URL https://www.microsoft.com/en-us/research/publication/analyzing-and-mitigating-data-stalls-in-dnn-training/. → page 15

[39] R. Montgomery. Applied statistics and probability for engineers. → page 33

[40] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi:10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646. → pages 9, 23

[41] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. URL https://arxiv.org/abs/1906.00091. → pages iii, xi, 1, 3, 4, 40, 41

[42] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal, K. Nair, I. Gao, B.-Y. Su, J. Yang, and M. Smelyanskiy. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *arXiv e-prints*, art. arXiv:2003.09518, Mar. 2020. → pages 1, 2

[43] Nvidia. NVIDIA Collective Communications Library (NCCL). https://docs.nvidia.com/deeplearning/nccl/index.html. → page 42

[44] Nvidia. Accelerating wide deep recommender inference on gpus, 2017. https://developer.nvidia.com/blog/accelerating-wide-deep-recommender-inference-on-gpus/. → page 4

[45] M. H. Nvidia Inc. Vinh Nguyen, Tomasz Grel. Optimizing the deep learning recommendation model on nvidia gpus.

https://developer.nvidia.com/blog/optimizing-dlrm-on-nvidia-gpus. → page 22

[46] L. M. E. O. I. O. Oleksii Hrinchuk, Valentin Khrulkov. Tensorized embedding layers for efficient model compression, 2020. URL https://arxiv.org/abs/1901.10787. → page 12

[47] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learnng. Oct. 2017. → page 9

[48] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017. → page 42

[49] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998. → page 39

[50] C. Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft. *Microsoft Blog*, 2019. → page 23

[51] H.-J. M. Shi, D. Mudigere, M. Naumov, and J. Yang. *Compositional Embeddings Using Complementary Partitions for Memory-Efficient Recommendation Systems*, page 165–175. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379984. URL https://doi.org/10.1145/3394486.3403059. → pages 17, 18

[52] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019. → page 23

[53] B. Smith and G. Linden. Two decades of recommender systems at amazon.com. *IEEE Internet Computing*, 21(3):12–18, 2017. doi:10.1109/MIC.2017.72. → page 1

[54] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981. ISSN 0001-0782. doi:10.1145/358699.358703. URL https://doi.org/10.1145/358699.358703. → page 10

[55] Y. Sun, F. Yuan, M. Yang, G. Wei, Z. Zhao, and D. Liu. A generic network compression framework for sequential recommender systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, page 1299–1308,

New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi:10.1145/3397271.3401125. URL https://doi.org/10.1145/3397271.3401125. → page 22

[56] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33, 2020. → page 24

[57] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. *SIGMETRICS Perform. Eval. Rev.*, 25(1):100–114, June 1997. ISSN 0163-5999. doi:10.1145/258623.258680. URL https://doi.org/10.1145/258623.258680. → page 10

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, 2017. → page 46

[59] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proc. VLDB Endow.*, 7 (11):1011–1022, July 2014. ISSN 2150-8097. doi:10.14778/2732967.2732976. URL https://doi.org/10.14778/2732967.2732976. → page 19

[60] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, Feb 2019. doi:10.1109/HPCA.2019.00048. → page 4

[61] X. Wu, H. Xu, H. Zhang, H. Chen, and J. Wang. Saec: similarity-aware embedding compression in recommendation systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 82–89, 2020. → page 22

[62] P.-F. Z. H. Y. Yang Li, Tong Chen. Lightweight self-attentive sequential recommendation, 2021. URL https://arxiv.org/abs/2108.11333. → page 15

[63] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems, 2020. → page 20

[64] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156, 2018. doi:10.1109/MASCOTS.2018.00023. → page 15