Accelerating Input Dispatching for Deep Learning Recommendation Models Training

by

Muhammad Adnan

BEE, National University of Sciences and Technology (Pakistan), 2013

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 Input Dispatching for Deep Learning Recommendation Models Training" submitted by Muhammad Adnan 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, UBC
Supervisor

Karthik Pattabiraman, Professor, Electrical and Computer Engineering, UBC
Supervisory Committee Member

Mohammad Shahrad, Assistant Professor, Electrical and Computer Engineering, UBC
Supervisory Committee Member

Abstract

Deep-Learning and Time-Series based recommendation models require copious amounts of compute for the deep learning part and large memory capacities for their embedding table portion. Training these models typically involves using GPUs to accelerate the deep learning phase but restrict the memory-intensive embedding tables to the CPUs. This causes data to be constantly transferred between the CPU and GPUs, which limits the overall throughput of the training process. This thesis offers a heterogeneous acceleration pipeline, called Hotline, by leveraging the insight that only a *small number* of embedding entries are *accessed frequently*, and can easily fit in a single GPU's local memory. Hotline aims to pipeline the training mini-batches by efficiently utilizing (1) the main memory for not-frequently accessed embeddings, (2) the GPUs' local memory for frequently accessed embeddings and their compute for the entire recommender model, whilst stitching their execution through a novel hardware accelerator that gathers required working parameters and dispatches training inputs.

Hotline accelerator processes multiple input mini-batches to collect and dispatch the ones that access the frequently-accessed embeddings directly to GPUs. For inputs that require infrequently accessed embeddings, Hotline hides the CPU-GPU transfer time by proactively obtaining them from the main memory. This enables the recommendation system training, for its entirety of mini-batches, to be performed on low-capacity high-throughput GPUs. Results on real-world datasets and recommender models shows that Hotline reduces the average training time by 3.45 in comparison to a XDL baseline when using 4 GPUs. Moreover, Hotline increases the overall training throughput to 20.8 epochs/hr in comparison to 5.3 epochs/hr for Criteo Terabyte dataset.

Lay Summary

Deep Learning Recommendation Models (DLRM) are becoming an important class of Deep Neural Network (DNN) workloads because of computational and memory requirements. Training such large models require the use of CPU to accommodate embedding tables and GPU to accelerate the compute intensive neural network. For real world datasets, up to 25% of total training time can be spent on transferring the embeddings between CPU and GPUs. In this thesis, we propose a heterogeneous acceleration pipeline that efficiently utilizes the compute and high bandwidth memory (HBM) of GPUs to reduce the data transfer between CPU and GPU. We designed a novel hardware accelerator that gathers required working parameters and processes them in a pipeline fashion to dispatch training inputs to the GPUs to increase the DLRM training throughput. We leverage frequently-accessed embeddings to optimize embedding data placement on GPU's limited memory, that results in speedup of overall training time without losing any accuracy.

Preface

This thesis is the result of work carried out by myself, under supervision of my advisor, Dr. Prashant Nair and mentorship of Dr. Divya Mahajan (Microsoft Inc.). All chapters are based on work submitted to The 49th International Symposium on Computer Architecture **(ISCA'22)**. We leverage idea of Frequently Accessed Embeddings (FAE) [6] based on work published in Proceedings of Very Large Data Bases **(VLDB'22)**. I was responsible for conceiving the ideas, designing and conducting the experiments, compiling the results, 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 with her knowledge and expertise due to her expertise in this area of research. She provided insights into the recommendation model training limitations, helped in designing experiments, and assisted with the tools used for these experiments.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit
DLRM	Deep Learning Recommendation Model
DNN	Deep Neural Network
EAL	Embedding Access Logger
EMB	Embedding
GPU	General Processing Unit
HBM	High Bandwidth Memory
LFU	Least Frequently Used
MLP	Multi-layer Perceptron
NCCL	NVIDIA Collective Communications Library
PCIe	Peripheral Component Interconnect Express
RM	Recommendation Model
RRPV	Re-reference Predictor Value
SRRIP	Static Re-reference Interval Predictor
TBSM	Time Based Sequence Model

Acknowledgments

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

In addition to my advisor, I would like to thank my thesis examining committee, Dr. Karthik Pattabiraman, and Dr. Mohammad Shahrad, for their insightful feedback and possible avenues to explore in this area.

I would like to thank Dr. Divya Mahajan and my other colleagues at UBC for their support and stimulating discussions throughout my research endeavors.

Finally, I would like to thank my family, specially my late father for providing me with unfailing encouragement and backing throughout my years of study. My academic achievements would not have been accomplished without their support.

Chapter 1

Introduction

Deep learning models [18, 44, 65] exhibit a high-degree of coarse-grained parallelism, thus employ multiple GPUs or accelerators for their distributed high-throughput training [12, 29, 43, 55]. Recommendation models [24, 70] are a popular class of these algorithms that aim to provide the user with personalized choices based on their past behaviour or other users. Recommendation models consume more than 50% of total AI training cycles [5]. In contrast to traditional Deep Neural Networks (DNNs), recommender models are a disaggregated conflation of compute-intensive neural networks and memory intensive embedding tables [28]. These embedding tables store the user and item features, and their size is anticipated to increase as more users and items interact; well beyond the anticipated improvements in general purpose platforms [20, 53]. For instance, a recent work from Baidu has shown that the size of production-scale recommendation models can be 10 TB in size [81].

Current state-of-the-art recommender model training utilizes a combination of CPUs and GPUs to train Deep Learning Recommendation Model (DLRM) [56] and Time-Based Sequence Model (TBSM) [31]. The CPU nodes offer high memory capacity for storing large embedding tables and training data. However, from compute standpoint, CPUs are highly inefficient and do not offer the necessary throughput. Hence, GPUs are used to achieve a high compute throughput for the neural network portion of the models by exploiting coarse-grained parallelism at a mini-batch level [34] by replicating neural network portion of each GPU.

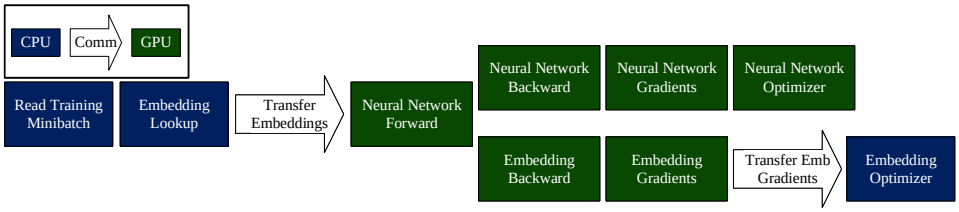

Figure 1.1: Baseline execution flow of a typical recommendation model. In this approach the entire embedding table is stored and processed on CPUs whereas the GPUs process the neural layers. Embeddings stored on CPU are transferred to GPU to be used by subsequent layers in the forward pass and embedding gradients are transferred back to CPU in the backward pass.

1.1 Motivation

Figure 1.1 shows the training flow of a recommender model in this hybrid execution mode with the CPU handling the embedding data and the GPUs performing execution of the DNN phase. This includes reading the training inputs, fetching corresponding embedding data from CPU main memory having low bandwidth, which results in inefficient embedding lookup and throttles system performance. After doing embedding lookup on CPU, respective embedding weights are transferred to GPU via PCIe x16 link. Neural network part of model is executed and complete interaction of neural network and embedding data is done on GPU devices in data parallel fashion. After completing the forward pass on GPU, backward pass is also executed on GPU as all the parameters are already available on GPU and gradients of neural network and embedding parameters are calculated. Gradients of neural network parameters are applied on GPU during optimization as neural network part of model is available on each GPU, but embedding gradients are sent back to CPU via PCIe x16 link for doing embedding optimization on CPU.

Figure 1.2 illustrates that, for four real-world datasets, up to 25% of the total training time can be spent on transferring the embedding weights during forward pass and embedding gradients after backward pass between CPU and GPUs. This number increases upto 45% for bigger models like in case of model used for Criteo Terabyte [16] dataset. Additionally, only neural network part of recommender

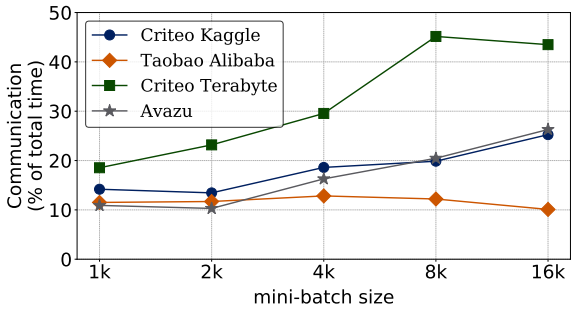

Figure 1.2: Percentage of training time spent on communication.

model is only stored on GPU main memory and GPU main memory is not used for embedding potion of model at all. If GPU main memory can be utilized for executing embedding lookup then high bandwidth GPU main memory can speedup training process. However, as the embedding entries cannot fit within limited GPU main memory, they are stored in the CPU memory. This implies that the baseline training process suffers from three main bottlenecks (1) frequent embedding data transfer between CPU and GPU via PCIe link, (2) storing the embeddings on limited bandwidth CPU main memory and (3) storing the embedding data far from computational units that is GPU.

1.2 Approach and Contributions

This thesis observes that the baseline execution fundamentally assumes that all embedding entries are equally important. However, as these models are used to provide personalized recommendations to users, semantically they deal with training inputs that exhibit high popularity [6, 13, 61]. This implies that certain inputs are *way more popular* than the others, which in-turn means that some embedding entries are accessed more frequently. We refer to these embedding entries as *frequently-accessed entries*. While the frequently-accessed embeddings are more critical from a compute perspective, recommender models do require other entries to enable diversity of recommendations.

Using this insight, this thesis offers a *throughput-optimized* heterogeneous acceleration pipeline, called Hotline, that mitigates the above-mentioned inefficiencies of training large-scale recommender models. Hotline aims to use GPUs to

execute the entire training core of the recommender model, i.e, forward pass, backward pass, loss function, and optimizer, due to high throughput. Hotline does so by determining which embeddings are frequently-accessed. It then stores these frequently-accessed embeddings locally on every GPU, i.e., close to the compute to reduce both the embedding data transfer time and also ensure high-throughput embedding computations on GPU. However this poses a challenge, because a mini-batch of inputs (2K-32K) can access a combination of frequently-accessed and not-frequently-accessed embeddings, albeit disproportionately towards frequently-accessed entries. As the GPU processes the entire mini-batch in parallel, obtaining several embedding entries which are not frequently-accessed from CPU main memory can become the bottleneck. To overcome this, Hotline offers a novel *accelerator* that sits between the GPU devices and the main memory, to rapidly access the not-frequently-accessed embeddings and the training data.

The Hotline accelerator at runtime, dynamically gathers the working parameters and schedules mini-batches onto the GPU. The accelerator processes and pipelines multiple mini-batches at a time to hide the embedding data transfer latency. The accelerator across its mini-batch window reforms the mini-batches into two classes. The first class of mini-batch only accesses frequently-accessed embeddings and are directly scheduled onto GPU. This is because the local memory of the GPUs contains the required working parameters. The second class of mini-batch contains the remaining inputs that access embeddings outside the frequently-accessed zone. To hide its data transfer latency, the accelerator gathers the required working parameters from CPU memory while the execution of the frequently-accessed inputs is performed on the GPUs. This enables the accelerator to schedule this mini-batch onto the GPU in a pipelined fashion. Thus, Hotline eliminates embedding data transfer latency for frequently-accessed-only training inputs and hides the communication latency for remaining inputs by efficiently pipelining and scheduling mini-batches.

Contributions: The main contributions of this thesis are:

- Identify challenges in training large recommender models and system bottlenecks.
- Leveraging the popularity of training inputs to infer that certain embedding

table entries are accessed significantly more than the others. During training, the local memory of GPUs can be used to store these frequently-accessed embeddings and this enables them to efficiently utilize the high-throughput GPU compute and GPU HBM.

- Providing a specialized accelerator design that dynamically determines the access skew within embeddings during the first training epoch and uses this information to efficiently pipeline the mini-batch dispatch onto GPU while concurrently obtaining not-frequently-accessed embedding working parameters from the CPU main memory.

- Offering Hotline, a technique that aims to maximize training throughput by stitching and pipelining the execution of recommender models into GPUs. Hotline uses an accelerator to quantitatively determine the embedding properties and schedule computations on GPUs. This enables Hotline to use GPUs to increase the throughput of the the entire training process whilst hiding the transfer latency to fetch working parameters for large models.

We evaluate Hotline across a wide range of publicly available deep learning and time-series based recommender models. We compare our end-to-end training results with the current state of art CPU-GPU baseline [36, 39] where CPUs store and compute on the embeddings and multiple GPUs parallelize the neural network computations across a mini-batch.
Our main results are as follows:

- Hotline maintains baseline training accuracy.

- The Hotline accelerated end-to-end training observes, on average, 3.45 speedup over the 4-GPU XDL baseline.

- Hotline's accelerator pipeline can process 20.8 epochs per hour in comparison to Intel optimized DLRM baseline that can process 5.3 epochs per hour for a training run that uses 16K mini-batches and 4-GPUs.

- Hotline increases the overall training throughput by 3 compared to Intel optimized DLRM baseline.

The rest of the thesis is organized as follows. We first give a brief background about the problem and tools used in Chapter 2, then we provide a brief overview of related work in Chapter 3. We identify the main challenges in Chapter 4 and present our Hotline system design in Chapter 5. Architecture of Hotline is being presented in Chapter 6. We evaluate our proposed techniques in Chapter 7. Finally, we conclude the thesis in Chapter 8 with future work.

Chapter 2

Background

2.1 Deep Learning Recommendation Model

Figure 2.1 showcases the general model structure of deep learning based recommender models. As the figure shows, on the one hand, these models comprise embedding tables that contain the categorical features for each user and item and impose a large memory footprint. On the other hand, the continuous features are processed directly by reasonably sized neural network layers which range from 10s of Multi-layer Perceptrons (MLP) to large deep learning based networks. Subsequently, using embedding data and initial neural layer outputs second-order interactions of different features is computed. Finally, the results are processed with another neural network and fed into an activation function to give a probability of a click. These models exhibit unique challenges as the embedding entries tend to not fit on GPU memories, but would benefit from using GPUs at handing neural network layers. Keeping these challenges in mind, we discuss the training process of these recommender models and how it differs from traditional DNN-only models.

2.2 Distributed Training Process

In the past few years, it has been established that training a production scale ML model requires multiple devices. Most commonly, data parallel [43] training is deployed, where a mini-batch of training inputs is processed in parallel across de-

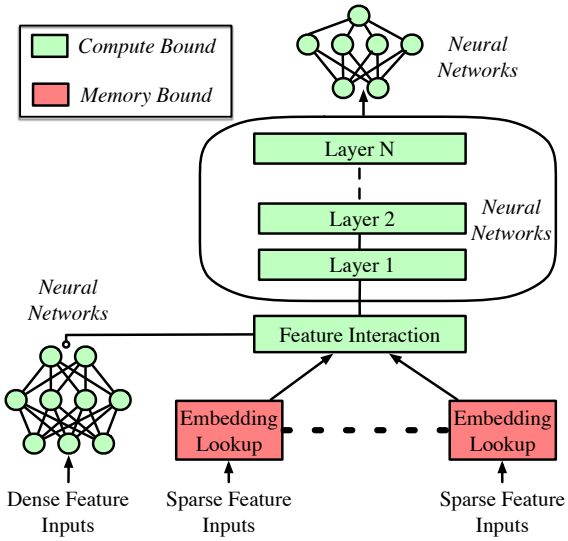

Figure 2.1: Typical recommender systems [27, 31, 56] model. Models comprise of an interconnection of compute-bound Neural Networks like DNNs and MLPs in tandem with the memory-bound (size and bandwidth) Embedding Tables.

vices and model parameter gradients are computed. Subsequently, these gradients are synchronized across the entire mini-batch and fed into the optimizer to obtain the final feature update. Data parallel training requires only one synchronization after all the devices have executed their portion of mini-batch. As the size of DNNs is increasing day by day, one single GPU or accelerator cannot handle the entire model, thus in addition to data parallel, model [12] and pipeline [29, 55] parallel modes are also being employed.

Model parallel and pipeline parallel currently split the model across devices for execution. Intermediate activations and results are transferred over the network to the next device in the pipeline. Such techniques work efficiently in the case where these splits are load-balanced [72] across devices to maximize throughput, i.e, each split often exhibits a high computational and memory intensity, thus utilizing resources effectively. This is because, the device-to-device, especially GPU-GPU communication costs for some of these models can be as high as 60% on average [2, 55, 74], especially if compute and communication are not overlapped. However, model parallel for recommender systems is inefficient use of resources as the embedding tables would have to be split across devices but they do not exhibit

Figure 2.2: Frequency of accesses per embedding for an entire training epoch. The frequently-accessed embeddings tend to have more than ten orders of magnitude more accesses as compared to not-frequently-accessed embeddings.

very high compute intensity. Using multiple GPUs just for memory capacity is not optimal, as it leads to low compute utilization and incurs a high communication overhead of collecting embedding parameters from all the devices. Instead, we dig in to the semantics of user behaviour, i.e. training input, consumed by recommender models to determine how we can ef ciently utilize the memory hierarchy and distribute training to maximize compute throughput.

2.3 Popularity in Training Inputs

An important property of recommendation system inputs is that certain users and items observe very high popularity. This implies that certain embedding entries are accessed more often that the others. Figure 2.2 shows that, across real-world datasets, a small number of embeddings constitute most of the access. The frequently-accessed embeddings have more than two orders of magnitude accesses as compared to not-frequently-accessed embeddings.

For real world datasets, frequently-accessed embeddings can cover >75% of training inputs. Even though frequently-accessed embeddings exhibit high temporal locality, the remainder of the entries contribute heavily towards improving the overall ef cacy of the model. Nonetheless, from a performance perspective, embedding entries that are frequently-accessed are critical as they are repeatedly used throughout the training process.

Chapter 3

Related Work

System level [29, 55] optimizations and architectural innovations [10, 11, 62] for training deep neural networks have gained widespread attention. Instead in this work we focus on optimizing training of recommender system models, as they commonly facilitate commercial applications including but not limited to social media [8, 9], e-commerce [47, 76], media streaming [75], etc. Prior work on recommender models with industry-scale datasets show that access patterns follow a Power or Zip an distribution [79] and thereby reaf rm our observation of frequently-accessed entries.

We classify related work into multiple broad categories. The rst category includes the techniques employed to predict user-item interaction that forms the basics of recommender models. The second category summarizes the placement of embedding parameters across the available memories and near-memory processing techniques for embedding tables. Next, we review some techniques suggested to reduce the memory intensity of recommender models. Following to this we present an overview of different machine learning accelerators employed to accelerate the compute intensive DNN workloads. Distributed machine learning training is being reviewed with different techniques to reduce the embedding dimension to tackle memory constraints. Finally, we discuss some software optimization techniques to optimize the recommender models execution on CPU.

3.1 Recommendation models and designs

In general, recommender models employ collaborative filtering [66] that makes predictions for users and items based on the preferences of other users and items. Matrix factorization technique is commonly used to decompose the user-item interaction to a lower dimensionality [42, 51]. Recently, there has been interest in deep learning based recommender models that use embedding tables [28] augmented with deep neural networks [31, 56, 80] to incorporate relevant query and item features. We focus on these models as they are commonly used in production and consume large number of compute cycles in data centres [26, 57]. Even with their popularity, much of the research has focused on optimizing the inference phase of these models [25, 26, 30, 46, 59]. Unlike inference, training is throughput optimized and tends to be more compute and memory intensive. Certain solutions that offer training optimizations [23, 68] or acceleration [40, 52, 62] do not maximize training throughput by facilitating better memory and bandwidth utilization of a distributed GPU system.

3.2 Embedding parameter placement

Work in [36, 81] offers a hierarchical parameter server to store the working data close to computation, i.e, GPU, at runtime. However, this work treats all embedding entries equally and requires continuous movement of embedding data between main memory and GPU. Work in [5], aims to understand the implications of different embedding table placements within an heterogeneous data-centre. Hotline delves into the access pattern of each dataset and uses this information at runtime to only obtain the non-frequently-accessed embedding entries from external memory. Works in [19] propose techniques to store embedding tables in non-volatile memories and allocate a certain portion of DRAM for caching. This work, however, does not support GPU based training executions with replicated hot embeddings and does not deal with perceptive input pre-processing to reduce the overhead of communication between devices. Recent work in [23, 40, 68] has also proposed solutions to accelerate near-memory processing for embedding tables, but do not facilitate distributed training of entire recommender models using GPUs.

3.3 Model optimizations to mitigate memory intensive training

Prior work [60], optimizes training by modifying the model either through mixed-precision training or eliminating rare categorical variables to reduce the embedding table size. Even with these optimizations real dataset's embedding table cannot fit on a GPU. Such approaches that change the data representation and/or embedding tables, require accuracy re-validation across a variety of models and datasets. Hotline on the other hand 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. More work has been done on compression [17, 32, 71, 78] and sparsity [21] to reduce the memory footprint of machine learning models in general. Hotline obtains better memory utilization of devices without employing overheads such as compression/decompression and sparse operations. Nevertheless, Hotline is orthogonal to these techniques and can be used in tandem with them to improve the memory efficiency even further.

3.4 General purpose machine learning accelerators

There has been a plethora of work on designing accelerators to execute compute portion of deep learning models [11, 14, 37, 45, 48, 52, 62, 64, 67]. Certain works have also offered accelerators for recommender models, especially collaborative filtering based models [52, 62]. Instead in this work, we do not aim to design a specialized architecture to optimize the compute of deep learning based recommender models. Hotline facilitates training by efficiently dispatching and gathering working parameters and input mini-batches to an already established GPU based system. GPUs are the go-to means to accelerate the deep learning models due to there close-knit integration with easy-to-use machine learning libraries [3, 4, 63]. Nevertheless, similar to GPU-based training, Hotline accelerator can be used to enhance the overall benefits of deep learning accelerators by pipelining their execution and gathering working parameters efficiently.

3.5 Distributed machine learning training

Distributed training has become the norm to train large scale models. Data parallel training [43] forms the most common form of distributed training as it only requires synchronization after the gradients generated in backward pass of training. However, as models become bigger and bigger [65, 69], model parallelism [12, 29] and pipeline parallelism [55] are becoming common as they split a single model onto multiple devices. Nonetheless, the techniques employed to automatically split the models [35, 72], offer model parallelism solutions to enable training of large model with size constrained by the accelerator memory capacity. Unlike DNNs, for recommender models, even though embedding tables are becoming larger, splitting them across devices just for memory capacity (especially GPUs) is sub-optimal and does not effectively utilize GPU resources. In this work, in contrast to past model parallelism work of splitting model, we use the fact that embedding table does not have to be fully replicated to perform data parallel training, instead only hot embedding entries can be sent to all the devices. Nonetheless, our optimization is orthogonal to these past techniques, as it can be employed in tandem with pipeline and model parallel distributed training to reduce the memory footprint further of embedding table per device.

3.6 Embedding dimension reduction of recommendation model sparse parameters

Another dimension explored by researchers to reduce the size of memory intensive embedding tables in recommender models is to use the optimal size for different embedding entries based on respective access frequencies. Some work [49] has offered a re-inforcement learning (RL) based automated way to search for optimal embedding size for each user and item entries during training process. ANother work [22] used multi-layer embedding training architecture that trains embeddings via a sequence of linear layers to derive superior embedding accuracy vs. model size trade-off. Although it is being claimed that model accuracy is being met using optimal dimension per embedding entry but this approach slows down the overall training process and reduces overall training throughput. While FlexShard leverages the access frequencies to move frequently-accessed embedding entries closer to

compute unit (GPU) without changing the embedding dimension size to avoid any accuracy loss. Also, Hotline utilizes the GPU HBM in a more efficient way for recommendation models training.

3.7 Software optimizations for recommendation models

Work in[39, 41] offers software based optimizations for executing recommender models on CPU. Work in [41], offers high-performance kernel library, from ground up to perform high-performance quantized inference on current generation CPUs but does not deal with training. Intel's work [39], optimizes the embedding lookup operator for receommender model training by parallelizing the embedding operation and using lock free updates.

3.8 Summary

Recommender systems account for most of AI cycles in cloud computing centers that has resulted in glut of research in understanding the complexities of recommender models. Prior work [26, 57] has done detailed characterization of recommender models to identify the system bottlenecks in recommender systems. To overcome system bottlenecks, multiple avenues has been explored by researchers are not limited to near-memory processing [40], parameter server approach [36, 81], optimizing recommender model [22, 49] and software based optimizations [39, 41]. This thesis analyzes the frequency of the embedding entries accessed and relates the behavior of access entries to real world access trends to find the frequently-accessed embedding entries. Technique proposed in this thesis accelerates the input dispatching based on frequently-accessed embedding entries to speed up the overall training process on heterogeneous platform.

Chapter 4

Challenges

The goal of this thesis is to maximize throughput and utilize memory and compute resources ef ciently across all platforms. Thus, based on the access pattern of embeddings and to best exploit their temporal locality, intuitively, we want to place them close to compute. Hence, we place all the frequently-accessed embeddings on every GPU in its local memory. Nonetheless even after doing so, translating the embeddding access skew to performance poses the following challenges:

4.1 Determining and storing the access skew

Although, we are aware that embeddings exhibit access skew, however for every training job – training input and model – we need to determine which embedding entries are actually frequently-accessed. To do so, we can either sample the training input before or determine this dynamically during runtime.

Solution To reduce the overhead of our solution, Hotline determines the mapping of frequently-accessed embeddings during the rst training epoch. It only needs to process about 5% of the training inputs for most of the real world datasets to determine which entries in the embedding tables are highly accessed. This requires ne-grained but parallel embedding index tracking (unsuitable for a GPU) across mini-batches, hence, Hotline offers a novel accelerator that takes sampled training inputs and identi es frequently-accessed embedding entries.

4.2 Runtime training with mini-batches

Training often employs a mini-batch of inputs; each mini-batch can require a mix of accesses between frequently-accessed and other embeddings, albeit distributed unevenly. This implies that for every mini-batch working parameters can be spread across main memory and GPU's local memory. Thus, working parameters need to collected and sent to the GPU before computation. Gathering working parameters using GPUs is non-ideal as GPUs are not only inef cient at collecting irregularly scattered embeddings in memory, but would also waste their precious compute cycles to perform an intricate ne-grained operation.

SolutionThus, we provide an immensely parallel accelerator that processes multiple input mini-batches. The accelerator is adept at generating mini-batches that only access frequently-accessed embeddings and dispatch them to GPUs and for the remaining inputs gather the embeddings from main memory.

Overall,Hotline offers a multi-purpose accelerator that during the short span of initial training determines the embedding access skew and subsequently uses this skew to dispatch and pipeline mini-batches onto the GPU.

4.3 Summary

In this chapter, we analyzed the challenges associated with increasing the through-put of DLRM training using frequently-accessed embeddings. Morover, how these challenges are related to training system bottleneck. We nd possible solutions to overcome these challenges to achieve desired goal.

Chapter 5

System Design

5.1 System Architecture

The baseline system is composed of a CPU that is connected to multiple GPU devices via the PCIe-link. The GPU devices are connected through a fast back-end interconnect NVLink [2] Figure 5.1 shows the high level system architecture with the CPU, GPU devices, and Hotline accelerator. Hotline enhances the baseline system by adding the accelerator on low profile Gen3 x16 PCIe slot as the accelerator needs to constantly access data that is stored in main memory. Adding Hotline accelerator does not change the total number of GPU devices connected to the system as it uses the extra available low profile PCIe slot. We place the Hotline accelerator on PCIe switch because the accelerator needs to fetch training inputs and non frequently-accessed embeddings. Accelerator has access to the DMA engine via PCIe switch and can directly access main memory whilst circumventing the high-overhead CPU pipeline entirely. Hotline does not require any architectural changes to CPU and GPU devices to be compatible with our accelerator but only requires an its own software stack to communicate with DMA engine and GPU device 0. Hotline altogether, trains in two phases described below.

Figure 5.1: Hotline's heterogeneous system architecture. The accelerator connects to low pro le Gen3 x16 PCIe slot and can directly access main memory via DMA enginer to obtain non-frequently-accessed embeddings to relay to the GPU.

5.2 Hotline Training Execution Pipeline

5.2.1 Access Learning Phase

In the rst phase, the accelerator for the rst few mini-batches only during the rst epoch of training, determines the embedding access skew. As Figure 5.2 shows, empirically, 4% of the inputs is suf cient to identify all the frequently-accessed embeddings, with high probability. Conservatively, we sample and process rst 5% of the mini-batches. 5% sampling rate works for most of the datasets like Criteo Kaggle [15] and Criteo Terabyte [16]. But for some datasets we need to sample more training inputs for more accurate learning. For Avazu [38] we sampled 15% of training inputs while for Taobao Alibaba [7] we sample 20% of training inputs as it contains time series data. Only the indices of the embedding entries that are accessed most number of times are stored in the accelerator to be used by the subsequent mini-batches. We call this the access learning phase. The details of how the accelerator performs this learning is described in Section 6.

Once the rst phase is completed, second phase starts in which recommender model is being trained on complete set of training data. We perform the access

Figure 5.2: Determining the mini-batch sampling rate.

learning on the accelerator because for the remainder of the training, the embedding indices which are frequently-accessed are required to classify and gather popular and non-popular inputs and into mini-batches. This constitutes the second phase of training, which we refer to as the *runtime accelerated phase*

5.2.2 Runtime Accelerated Phase

During this phase of execution, the Hotline accelerator operates on a set of mini-batches, called its working set. Based on the capacity of the accelerator and the size of each mini-batch, varying number of mini-batches can be processed in parallel. In this phase, the accelerator trains the recommendation model in a pipeline fashion by classifying popular and non-popular inputs across its working set into mini-batches. The inputs that only accesses the popular embeddings are gathered together as a mini-batch and directly scheduled onto the GPUs as its local memory contains the required frequently-accessed embeddings. In the meantime, when GPUs are executing on popular inputs, the accelerator re-processes the non-popular inputs and gathers its respective embeddings from CPU and GPU. This enables the Hotline accelerator to hide the embedding gathering latency for inputs which access embeddings available in main memory.

Execution pipeline example.

Figure 5.3 shows an example execution pipeline of Hotline. In this example the working set of the Hotline accelerator is four mini-batches. All the training inputs are obtained by the accelerator from the main memory. Four minibatches of 16 k inputs, for a large dataset such as Criteo Terabyte [16] amount to 10 MB of data.

Figure 5.3: An example execution pipeline of Hotline accelerated training. Hotline accelerator processes multiple mini-batches to classify popular and non-popular inputs, schedule the popular mini-batch directly on GPU, and in meantime gather working parameters for non-popular mini-batch to schedule onto GPU. Accelerator hides the data gather latency behind GPU execution.

We observe that even a working set of four mini-batches is enough to hide the latency of obtaining working embedding parameters from main memory for the non-popular inputs. This is because ≈75% of the inputs are popular, hence in our example approximately 3 out of the 4 minibatches would contain only popular inputs and can be scheduled onto the GPU directly. Section 7 offers empirical evidence that this working set is sufficient to overlap the data transfer latency for the 1 non-popular minibatch. The non-popular mini-batch requires embeddings from both frequently-accessed and non-frequently-accessed embeddings. frequently-accessed embeddings are only available on GPU devices and non-frequently-accessed embeddings are available on CPU main memory. Hotline accelerator gathers required embeddings from respective devices and schedules the non-popular inputs to GPU once the GPU becomes available.

For popular mini-batches both the forward and backward pass for the entirety of the recommender model, shown in Figure 1.1, executes on GPUs as the required parameters are in GPU's local memory. For non-popular mini-batches, non-popular embeddings are gathered from CPU main memory via DMA engine during the forward pass and sent to GPU. During the backward pass, the non-frequently-accessed updated embeddings are sent to Hotline accelerator where they gets updated in CPU main-memory via DMA engine.

As such, if the execution was entirely sequential, the non-popular embedding preprocessing and gathering would be in the critical path; but the Hotline accelerator creates a pipeline execution to hide this latency under popular mini-batch execution and reduces the overall communication time for popular mini-batches.

5.3 Summary

In this chapter, we present the overall system design of Hotline based DLRM training. Making architectural changes in currently available AI servers to speed up the training process is not openly accepted by industry as that will have huge cost factor. We designed our Hotline system in such a way that it won't have any effect on architecture of current AI servers and will be able to speed up the whole training process with minimal additional cost. Hotline accelerator is placed on low pro le Gen3 x16 PCIe slot instead of normal PCIe slot to keep the same number of GPU devices and computational power of AI server.

Chapter 6

Accelerator Architecture

This section describes the detailed design, architecture and software stack of the Hotline accelerator along with a data flow example. The goal of the Hotline accelerator is to enhance the GPU compute and throughput by (1) identifying frequently-accessed embeddings during the very initial phase of training and (2) intelligent schedule mini-batches for remainder of the process. The Hotline accelerator comprises seven main components shown in Figure 6.1.

The *hotline controller* is responsible for the execution flow of the accelerator. *Data dispatcher* streams inputs and issues relevant mini-batches to the *input eDRAM* and *lookup engine* It is also responsible to obtain the required input mini-batches and embedding entries from CPU and GPU. *Embedding Access Logger (EAL)* is populated with the frequently-accessed embedding indices during the access learning phase. This logger only needs to store the embedding indices (and not the data) because the accelerator does not perform any computation, rather is a structure for gathering and dispatching working parameters and input mini-batches. The logger becomes *read-only* after it has learned the indices of frequently-accessed embeddings; during the runtime accelerated phase. *Lookup engine* is an entirely parallel array of lookup engines that is responsible for all the logic that is performed to look into the Embedding access logger. *Input eDRAM* is double sided queue used to store the working set of mini-batches. *Scheduler* is used to organize the timeline to issue mini-batches to the GPU, in the popular and not-popular order per working set. *Reducer* contains array of arithmetic logic

Figure 6.1: Block diagram of Hotline accelerator. Controller is responsible for controlling the overall execution flow and data dispatcher collects and transfer all the required data.

units required to do element wise embedding pooling operation (i.e, sparse length sum) for the embedding weights (Non-popular embedding weights) coming from CPU main memory via DMA Engine in forward pass. Finally, Embedding eDRAM stores the reduced embedding vector, that comprises of embedding weights from CPU main memory and GPU HBM during forward pass. While during the backward pass, it stores the updated non-frequently-accessed embedding weights.

6.1 Hotline Controller and Data Dispatcher

Hotline controller and data dispatcher comprises two state machines, each for access learning and runtime accelerated phase. During access learning phase, the data dispatcher gets input mini-batches from CPU and sends them to the lookup engine which in-turn updates the EAL. Once the embedding logger is learned, during the next phase the data dispatcher directs each input in the mini-batch to the lookup engine to tag whether embedding indices are frequently-accessed or not.

The input classifier block in the controller receives the inputs back from lookup engine. If all the embedding indices are hit in the embedding access logger, then the input is tagged as popular input by the controller and sent to the front end of the input eDRAM. For non-popular inputs, the controller adds them to the other end of the input eDRAM, through the lookup engine and input classifier block. On the

Figure 6.2: Block diagram of Memory Controller.

side, the controller also awaits the scheduler to signal mini-batch dispatch. Once a mini-batch worth of popular inputs is collected in the input eDRAM, the scheduler sets the dispatch register, constantly polled by the controller. While the popular mini-batches are being dispatched to GPU, the controller reissues the non-popular mini-batch to the lookup engine from the input eDRAM to determine its working parameters. For non-popular mini-batch dispatched to lookup engine, contains mixture of frequently-accessed embeddings and normal embeddings. Input classi er creates a boolean mapping of frequently-accessed embeddings and normal embeddings and send the non-popular mini-bacth to the memory controller.

6.1.1 Memory Controller

Memory controller only handles the non-popular mini-batches in a pipeline while popular mini-batches are being dispatched directly to GPUs. For normal embeddings within non-popular mini-batch, memory controller uses the Address Reg and input embedding index to calculate the address of required embedding row in CPU main memory. Figure 6.2 shows that memory controller performs simple address generation for normal embeddings and dispatches Direct Memory Access (DMA) request to DMA engine along with calculated address and no of bytes to read based on the model sparse dimension. For frequently-accessed embeddings, memory controller simply dispatches the inputs to GPU device 0 for embedding weights present within frequently-accessed embedding table on each GPU device.

Address Registercontains the base address of each embedding table within the CPU main memory. During boot time, address register is being populated by CPU with the base address of each embedding table.

6.2 Embedding Access Logger (EAL)

Figure 6.3: Design of the embedding access logger.

The Embedding Access Logger maintains a log of the frequently-accessed embedding indices. The logger only stores the indices and not the embedding data. Figure 6.3 shows our design of the logger. Logger is written during the access learning phase and only reads during the runtime accelerated phase. This block is managed by a controller and is accessed using an n-entry queue. To improve parallelism, the embedding access logger (EAL) is split into m-banks with each bank containing multiple entries. The controller schedules multiple requests across banks in each iteration to achieve parallelism.

Figure 6.4: Impact of logger queue size on requests issued.

Figure 6.4 shows the average number of requests issued as the number of banks (n) increase and the input queue size (z) varies. We observe that a 512-sized queue with 64-banks helps achieve 60 parallel requests into unique banks per iteration.

Each logger entry comprises a 14-bit identifier, one valid-bit, and two access-count bits. During the access learning phase, for each input, based on which embedding indices it accesses, the counters are updated. To reduce conflicts between multiple embeddings being mapped to the same logger entry, we use a 4-way set-associative structure with an optimal replacement policy. Fig 6.5 shows the hardware architecture of our 4-way set-associative cache and how the access counter is being updated for each cache line.

Figure 6.5: Embedding Access Logger Cache Architecture.

Ideally, to fill the logger, we need to process the entire training data, log the accesses of each embedding entry, sort the entries based on the access frequency, and select the top entries. To avoid this huge pre-processing overhead, we offer a logger that is enhanced with a smart Least Frequently Used (LFU) replacement policy. We use a static Re-reference Interval Predictor (SRRIP) replacement policy with 2-bit Re-reference Predictor Value (RRPV) counter [33]. As frequently-accessed embeddings have a two orders of magnitude higher frequency of access, even a

2-bit RRPV counter can capture them. Figure 6.6 compares SRRIP based smart logger to the high-overhead oracle. Oracle can perfectly capture all the frequently-accessed indices. We observe that our embedding access logger with SRRIP based cache replacement policy observes a close to ideal hit rate.

Figure 6.6: Hotness captured by SSRIP.

After the access learning phase, the logger entries are made read-only. During the runtime accelerated phase, the logger only needs to provide whether a training input embedding index is frequently-accessed or not.

6.3 Lookup Engine

Lookup engine is a parallel 2D network of lookup engines that are responsible for looking up on every training input within EAL. Lookup engine, based on the information within an input, extracts which embedding entries that the input requires. This enables the lookup engine to send relevant information to update the embedding logger during the access learning phase. During the runtime accelerated phase, this information enables the lookup engine to tag inputs with a set of binary stream which speci es whether embedding accesses were hit or a miss in the logger. We offer an array of lookup engines because the operation to determine whether an input is popular are independent and can be parallelized, as several embedding entries are accessed per input. For instance, if an input requires 26 distinct embedding tables, the accelerator can achieve 26x throughput per input. The array enables exploiting parallelism within a training input (through different embedding indices) and across training inputs within the mini-batch working set. Figure 6.7 shows the innards of a single lookup engine. Each lookup engine consists of embedding table number, embedding index as input registers, a randomizer, update Index block and hot embedding index as output register. The input-output regis-

Figure 6.7: Lookup engine architecture.

ters capture the input to and from the controller. The embedding table number and embedding index for the input are merged and sent into a randomizer unit. The randomizer creates a hash of the (Embedding Index, Embedding Table) tuple.

Randomization enables the embedding logger to capture more frequently accessed entries as shown in Figure 6.8. Without randomization more inputs collide for the same logger entry. This causes the logger to trash out the colliding frequently-accessed embedding entries, thereby logging few (or none) of them. Randomization scatters the input access patterns to different logger entries. Specially for Taobao Alibaba [7] dataset, it has been observed that without randomization we get 0% inputs that completely access frequently-accessed embeddings. This is result of time based sequential data within a single input, each time based data evicts previous time slot embeddings access and embedding access logger was not able to store the embedding access for all time slots within a single input. Randomization scatters even time series embedding entries to different embedding logger sets which avoids eviction and gives good results. This increases the percentage of frequently-accessed embeddings that are captured during the access learning phase. We implement our randomizer block using a low-latency Fiestal Network implementation which incurs a 2-cycle randomization delay [50].

Figure 6.8: Effect of randomization on percentage of inputs captured.

6.4 Input eDRAM

Input eDRAM is a double ended queue that stores the entire working set of the training mini-batches. Inputs in the input eDRAM go through the lookup engine to be segregated into popular and non-popular mini-batch sets. The front end of the queue stores the popular inputs whereas the rear end stores the non-popular inputs.

6.5 Scheduler

For each working mini-batch set, once a mini-batch worth of inputs are collected in the front end of the input eDRAM, the scheduler sets the ready to schedule register and let data dispatcher take care of mini-batch. These inputs are sent directly to GPU devices for training by data dispatcher. As the popular mini-batches are being scheduled, the scheduler waits for all the non-popular inputs to be gathered in the rear end of the input eDRAM. For these inputs, controller sent them to memory controller to obtain the working parameters from GPUs for all the embedding indices present in the logger and obtains the remaining working parameters from main memory via DMA engine. Scheduler again issues the ready to schedule once the working parameters are obtained.

6.6 Reducer

As the working parameters required from CPU main-memory are accessed via DMA engine to reduce the access latency and minimize involvement of CPU processor, an embedding reduction array is required on the accelerator to reduce the embedding vectors into a single vector within an embedding table for

Figure 6.9: Embedding Reduction Array.

given set of input (i.e, sparse lenght sum operation). Reducer contains an an array of simple Arithmetic Logic Units (ALUs) to do element wise reduction operation in the forward pass of training. For each embedding table the reduced embedding vector is being stored within Embeddings eDRAM.

6.7 Embeddings eDRAM

To store the working parameters for non-popular mini-batch, a storage element is required where reduced embedding weights can be stored in the forward pass and then dispatched to GPU device via data dispatcher. Also after the backward pass, updated embedding weights needs to be stored for non-frequently-accessed embeddings. Embedding eDRAM is a storage memory for temporarily storing the working parameters in forward and backward pass before they have been collected for complete mini-batch and are being dispatched.

6.8 Hotline Instruction Set Architecture

Table 6.1: Hotline Instructions Set

Instruction	Operand 1	Operand 2	Description
dma_rd(op1, op2)	mem start idx	# bytes	DMA read request
dma_wr(op1, op2)	mem start idx	# bytes	DMA write request
v_add(op1)	input vector	-	point wise vector addition
v_mul(op1)	input vector	-	point wise vector dot product
s_wr(op1, op2)	addr reg idx	base addr val	write to addr reg
gpu_rd(op1)	sparse idx	-	read emb idx from GPU

Figure 6.10: Data ow in Hotline accelerator. Scheduler schedules mini-bacth 1,2 and 3 directly to GPU device while mini-batch 4 is being scheduled in parallel to Hotline accelerator for gathering working set parameters.

As Hotline acts as a bridge between CPU main memory and GPU devices and interacts with both CPU main memory and GPU device 0 via PCIe link, it requires a brief set of instructions to achieve required working set parameters. Hotline instruction set architecture consists of brief set of instructions to communicate with DMA engine and GPU device 0. As embedding sparse dimension is usually of the order of 16 (16-wide vectors to 128-wide vectors), single embedding weight vector size ranges from 64 bytes to 512 bytes. Each DMA read and write instruction requires maximum 64 bytes to 512 bytes. Table 6.1 shows brief overview of two Hotline instructions set required to communicate with CPU main memory via DMA engine and GPU device 0. Other than communication, other operations are being handled by hardware components within Hotline architecture and control signals are being generated by two state machine within Hotline controller.

6.9 Hotline Data Flow

Figure 6.10 shows a working example of Hotline accelerator. Input eDRAM contains set of four mini-batches with three being popular and one being not-popular. Scheduler schedules the rst popular mini-batch to GPU where popular mini-batch is being executed on GPU in data parallel fashion as all model parameters for popular mini-batch are present in local memory of each GPU. In parallel, scheduler schedules a non-popular mini-batch within input eDRAM to lookup engine to identify the frequently-accessed and non-frequently-accessed embeddings within each input of mini-batch. Input classi er within Hotline controller generates a Boolean

mapping of respective embeddings and dispatches non-popular mini-batch inputs sparse indices and boolean mapping to memory controller. Memory controller utilizes address generation register to find the memory address of respective sparse input index in CPU main memory for non-frequently-accessed embeddings. DMA read request is being sent to DMA engine for non-frequently-accessed embeddings and GPU read request to GPU device 0 for frequently-accessed embeddings. Working parameters from DMA engine and GPU device are fed to reducer block where embeddings are reduced into a single vector and stored in embedding eDRAM. Working parameters stored in embedding eDRAM are transferred to GPU device once non-popular mini-batch is being scheduled to GPU device after executing popular mini-batches. After backward pass, updated non-frequently-accessed embeddings are transferred back to Hotline accelerator from where they are being updated on CPU main memory using DMA engine.

6.10 Summary

In this chapter, we present the architectural details of each component within Hotline accelerator. How Hotline architecture is able to improve the overall training efficiency of the entire system without bogging the GPU down with fine grained but parallelizable tasks of segregating and collecting mini-batches and embeddings.

Chapter 7

Evaluation

We evaluate Hotline's overall training process, end-to-end performance gains, and perform architectural sweeps to thoroughly assess our accelerator design choices.

7.1 Experimental Setup

7.1.1 Recommender Models

Table 7.1 illustrates the specifications of four well-established diverse recommender models which are used to evaluate Hotline. The number of sparse parameters varies from 5.1M (RM1) to 266M (RM3). The number of dense parameters is around 5 orders of magnitude less than the number of sparse parameters. The bottom MLP computes on dense inputs, whereas the top MLP generates the final recommendation for the user. RM1 is a neural network dominated model, which in addition to the top and bottom MLP layers also uses a deep learning attention layer. RM2, RM3 models, instead, are embedding dominated as they consist of more sparse features and bigger embedding tables. RM4 is an average model with average size of dense neural network and sparse embedding tables. We use open source implementation of Deep Learning Recommendation System (DLRM) [56] and Time-based Sequence Model for Personalizing and Recommendation Systems (TBSM) [31] benchmarks to train the above mentioned models. TBSM trains RM1 model and DLRM is used to train RM2, RM3 and RM4 models. TBSM consists of an

Table 7.1: Recommender Model Architecture and Parameters

Model	Dataset	Features		Parameters			Neural Network Con guration			Size
		Dense	Sparse	Dense	Sparse	Sparse Dim	Bottom MLP	Top MLP	DNN	(GB)
RM1	Taobao Alibaba [7]	1	3	7.3 k	5.1 M	16	1-16	30-60-1	Attn. Layer	0.3
RM2	Criteo Kaggle [15]	13	26	287.5 k	33.8 M	16	13-512-256-64-16	512-256-1	-	2
RM3	Criteo Terabyte [16]	13	26	549.1 k	266 M	64	13-512-256-64	512-512-256-1	-	63
RM4	Avazu [38]	1	21	281.4 k	9.3 M	16	1-512-256-64-16	512-256-1	-	0.55

embedding layer and a time-series layer. The embedding layer within TBSM is implemented using DLRM and the TSL layer resembles an attention mechanism and contains its own neural networks. Attention MLP takes input from underline DLRM layer in a time series fashion and passes them through a nal MLP to provide a probability of a click.

7.1.2 Datasets

We use four real-world datasets to train the above mentioned models. Table 7.2 shows the detailed speci cation of each dataset. Criteo Kaggle [15] dataset has been taken from Display Advertising Challenge and contains advertising data from Criteo to capture user's preference by predicting click-through rate. Criteo Kaggle is being used to train RM2 model. Taobao Alibaba [7] is a user behaviors dataset from Taobao, for recommendation problem with implicit feedback offered by Alibaba. It contains 1M users, 100M interactions with 4.1M items. Taobao Alibaba dataset is being used to train RM1 model. Criteo Terabyte [16] is the largest publicly available dataset for users click logs. It is being used to train RM3 model.

Table 7.2: Dataset Speci cations

Dataset	Training Inputs	Size (GB)	Time Series Samples
Criteo Kaggle [15]	45 M	2.5	1
Taobao Alibaba [7]	10 M	1	21
Criteo Terabyte [16]	80 M	45	1
Avazu [38]	32 M	2.4	1

7.1.3 Software libraries and setup

DLRM and TBSM are con gured using Pytorch-1.9 and executed using Python-3. With Pytorch, we use the torch.distributed backend to support scalable distributed training and performance optimization [63]. For all the backend GPU-

to-GPU communication, the highly fast NVIDIA Collective Communication Library (NCCL) [58] is used. We compare Hotline training performance against two state of the art baselines: (1) DLRM and TBSM implementation on XDL [36] and (2) DLRM and TBSM implementation using intel-optimized PCL embedding bag [39]. For XDL implementation we use Tensor ow-1.2 [4] as computation backend. The baseline and Hotline executions use gather, scatter, and all-reduce collective calls offered using the high speed NVLink-2.0 interconnect.

7.1.4 Server Architecture

Table 7.3: System Speci cations

Device	Architecture	Memory	Storage
CPU	Intel Xeon Silver 4116 (2.1 GHz)	192 GB DDR4 (2.7 GB/s)	1.9 TB NVMe SSD
GPU	Nvidia Tesla V100 (1.2 GHz)	16 GB HBM-2.0 (900 GB/s)	-

The details about the server are provided in Table 7.3. They comprise 24-core Intel Xeon Silver 4116 (2.1 GHz) processor with Skylake architecture. Each server is connected to 4 NVIDIA Tesla-V100 GPUs, each with 16GB memory capacity. Every GPU and Hotline accelerator is communicating with the rest of the system via a 16x PCIe Gen3 bus. 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 by using the RDMA technology for directly accessing GPU HBM present in other nodes connected via network card.

7.1.5 Execution time measurements.

To obtain the execution time for CPU and GPU executions we measure the wall clock time to train the entire model based on their convergence criterion. For the accelerator runtime estimates, we use a custom cycle-accurate simulator based on speci cations in Table 7.4.

From our empirical studies, we observe that a size of 10 MB for the input eDRAM and 16 MB for embeddings eDRAM is suf cient as 10 MB input eDRAM can store at least 4 of the 16 k input minibatchs and 16 MB embeddings eDRAM

Table 7.4: Accelerator Specifications

Parameters	Settings	Parameters	Settings
Frequency	350 MHz	PCIe bandwidth	15.75 GB/s
EAL size	8 MB	No of Lookup Engines	64
Input eDRAM size	10 MB	No of Reducer ALU Units	16
Embedding eDRAM size	16 MB	-	-
Total Area	18.31 mm^2	-	-
Average Power	14 W	-	-

can store non-frequently-accessed embeddings for our largest dataset Terabyte. We used eDRAM [73] technology for Input eDRAM because of high storage density as compared to SRAM technology. Although one of drawback of eDRAM technology is destructive reads, but in our case we just need to read an input and embedding once so this drawback turns out to be beneficial for us. The embedding access logger size is capped at 8 MB as this only needs to capture the indices of the frequently-accessed embeddings. SRAM technology is used for embedding access logger due to its low access latency and non-destructive reads.

7.1.6 Area/Energy measurements.

For the Hotline accelerator, Cacti [54] is used to estimate the area/energy of the memory components using 45nm technology. All other components of Hotline accelerator were implemented in Verilog RTL and synthesized using Synopsis DC in the 45nm FreePDK process.

7.2 Results

7.2.1 Training Accuracy

Hotline pipeline reshuffles the mini-batches into two categories - popular and non-popular mini-batches, although only within its working set in the input eDRAM. This shuffles the inputs during training, hence, we evaluate the accuracy implications of Hotline. Figure 7.1 illustrates the Area Under Curve (AUC) accuracy metric for Criteo Kaggle, Criteo Terabyte and Avazu dataset as established by MLPerf [1, 77] community. For Taobao dataset, we use the test accuracy as

(a) Criteo Kaggle

(b) Taobao Alibaba

(c) Criteo Terabyte

(d) Avazu

Figure 7.1: Testing AUC for both baseline and Hotline heterogeneous acceleration pipeline.

performance metric, as AUC accuracy metric is not offered for TBSM model. Full-precision DLRM and TBSM models implementation using Intel's optimized DLRM [39] is used as baseline for accuracy comparison.

Table 7.5: Accuracy Metric Comparisons

Dataset	DLRM			Hotline		
	Accuracy (%)	AUC	Logloss	Accuracy (%)	AUC	Logloss
Criteo Kaggle	78.64	0.798	0.456	78.64	0.798	0.456
Taobao Alibaba	89.11	-	0.270	89.34	-	0.264
Criteo Terabyte	81.20	0.792	0.421	81.21	0.792	0.421
Avazu	83.61	0.766	0.387	83.61	0.768	0.386

We observe that Hotline completely follows the baseline test and train accuracy

Figure 7.2: The performance of Kaggle, Taobao Alibaba, and Terabyte training with Hotline vs XDL and optimized DLRM as baseline. All values are normalized to a 1-GPU XDL system.

and has virtually no impact on the ef cacy of the model. Table 7.5 compares the testing accuracy, AUC and cross-entropy loss for all workloads.

7.2.2 End-to-End Performance

Performance Bene ts

Figure 7.2 demonstrates the end-to-end performance bene ts obtained due to Hotline with varying number of GPUs. All the numbers are normalized to XDL 1-GPU setup. We deploy the commonly utilized weak scaling technique to conduct the experiments in which we scale the mini-batch size with number of GPUs, i.e., 1K for 1-GPU setup, 2K for 2-GPU setup, and 4K for 4-GPU across all the evaluated datasets. As the Figure 7.2 shows, Hotline unanimously, across all the models, datasets, and settings, reduces the overall training time. This is because, it is performing the entire recommender model compute on the GPU. XDL baseline uses a parameter server approach in which GPU is used to improve the ef ciency of Advanced Model Server by using a faster embedding dictionary lookup on GPU while CPU is used as backend workers. While optimized DLRM baseline execute embeddings on CPU and neural networks of GPU. Thats why we can see performance improvement of optimized DLRM baseline over XDL baseline. Hotline reduces the communication bottleneck of transferring embeddings across CPU and GPU and places the frequently-accessed embeddings closer to compute unit (GPU local memory) and gets performance improvement.

Figure 7.3: Latency breakdown 1, 2, and 4 GPU baselines and Hotline executions.

On average Hotline gives a speedup of 2.86 and 2.86 with 1-GPU executions, 3.4 and 3.3 with 2-GPU exeecution, 3.45 and 2.3 for 4-GPU execution over XDL and optimized DLRM baseline.

Latency Breakdown

Figure 7.3 shows the latency breakdown for baselines and Hotline execution for all workloads. As the figure shows, Hotline incurs a negligible overhead. For Criteo based datasets, Kaggle and Terabyte, which are more embedding and memory intensive, Hotline reduces the CPU to GPU communication time more significantly. For Taoboa dataset which is neural network centric, the deep learning execution overshadows the communication time. For Avazu dataset communication time is average. Nonetheless, in this case as well, Hotline is able to proportionally reduce the compute time, as Hotline enables most of the computation, especially the optimizer, to be performed on GPU, unlike the baseline. Hence, Hotline works well for both embedding heavy models by reducing and pipelining CPU to GPU communication time, and also for models which have a large deep learning layer by offloading more compute onto the GPU. With the continual advances in deep learning, it is expected that most of the recommender models will move towards utilizing larger neural networks, thus Hotline becomes even more critical as it utilizes GPU's local memory, its compute resources, and system bandwidth, efficiently.

Hotline Throughput Improvements

Hotline improves the overall throughput by processing a pipeline of input mini-batches. As Figure 7.4 shows, for a 4-GPU system, Hotline accelerated pipeline can process, on average, 3× more epochs/hour as compared to optimized DLRM baseline. Hotline throughput improves at a higher rate with increasing mini-batch than the baseline, because with a larger mini-batch, Hotline provides more compute opportunities for the GPUs to parallelize on.

Figure 7.4: Varying throughput with 4-GPU executions.

CPU to GPU Embedding Transfer Latency

Table 7.6 shows the absolute time spent by baseline and Hotline to perform CPU to GPU embedding data transfers. Hotline reduces the data transfer time, on average (4-GPU), by 2.65 ×, as popular inputs do not need to obtain embeddings from main memory. Moreover, Hotline hides this latency for non-popular inputs underneath the popular-minibatch GPU executions.

Table 7.6: CPU GPU data transfer time - 10 Epochs (mins)

Dataset	1-GPU		2-GPU		4-GPU	
	DLRM	Hotline	DLRM	Hotline	DLRM	Hotline
Criteo Kaggle	17.04	5.27	17.91	6.79	16.89	10.51
Taobao Alibaba	15.65	4.81	15.21	3.30	11.71	3.10
Criteo Terabyte	62.47	23.65	54.27	20.65	47.27	16.78
Avazu	11.96	3.41	14.71	3.33	11.51	4.78

Varying Number of Inputs in a Mini-batch

We also evaluate the end-to-end performance with varying number of inputs that constitute a mini-batch. Figure 7.5 shows the performance benefits of Hotline further accentuate with larger mini-batch sizes. This is due to two main reasons, larger mini-batch amortizes the overheads of Hotline accelerator across a larger working set and the scheduler also has to issue input dispatch commands fewer times. Secondly, larger mini-batches offer more opportunities for parallelism, and Hotline enables more compute (optimizer) to be performed on the GPUs during the whole training process, hence obtains higher benefits with increasing mini-batches.

Figure 7.5: Hotline speedup with varying mini-batch sizes.

7.3 Hotline Architectural Evaluation

7.3.1 Mini-batch Working Set

We compare the impact of the mini-batch working set on the execution time. For a small working set, e.g. 1, it implies that if there is even a single input which

accesses non-frequently-accessed embeddings, the accelerator will have to gather the working parameters for the whole mini-batch. Moreover, this also implies that this data transfer latency cannot be hidden behind popular input execution, thus leading to the a bottle-necked system. We observe that for larger datasets, a 4 mini-batch working set is suf cient to completely ll the pipeline and extract maximum throughput by increasing GPU utilization.

Figure 7.6: Varying minibatch working set in Input eDRAM.

7.3.2 Embedding Access Logger Size

Figure 7.7 shows the percentage of inputs accessing frequently-accessed embeddings captured by varying the logger size. As expected, for Criteoand Avazu datasets with more skewness, even a logger as small as 2 MB can capture most of the frequently-accessed indices. However for Taobao dataset that has a non-exponential skew, a very small logger cannot capture the access pattern, but once we increase the size beyond 8 MB we start to observe diminishing returns.

Figure 7.7: Percentage of inputs accessing frequently-accessed embeddings captured with varying logger size.

7.3.3 Power Breakdown

Figure 7.8 shows the power consumption across different components of the Hotline accelerator for a single training epoch of each dataset. Embedding eDRAM consumes the most power as it is the biggest memory element within our architecture and has comparatively higher power leakage. Embedding Access Logger and Input eDRAM have almost similar power consumption but varies across different recommendation models and associated datasets. Finally, the lookup engine consumes the least power as it is not a memory component and doesn't have memory power leakage associated with it. Currently, reducer is not utilized in our experiments as all datasets have single embedding lookup per embedding table.

Figure 7.8: Power Consumption of Hotline accelerator components.

7.4 Summary

In this chapter, we evaluate the accuracy and performance of Hotline against two state of the art baselines. We observe that Hotline speeds up the overall training time by 3.45 in comparison to a XDL baseline when using 4 GPUs without any loss of accuracy. Moreover, Hotline increases the overall training throughput to 20.8 epochs/hr in comparison to 5.3 epochs/hr for largest dataset i.e. Criteo Terabyte dataset using 4 GPUs execution.

Chapter 8

Conclusion and Future Work

8.1 Summary

Production-scale recommendation models deploy deep neural networks along with capacity and bandwidth-constrained embedding tables. The disparity between the requirements of these components requires a unique approach to improve the overall training throughput. In this paper, we present Hotline, a heterogeneous acceleration pipeline based on the insight that small portion of embedding tables are more frequently accessed in comparison to others. This offers a window of opportunity to place frequently-accessed working parameters close to the compute, i.e, on every GPU. Then, Hotline aims to dispatch popular inputs that only access these frequently-accessed embeddings directly to GPU, meanwhile obtaining the non-frequently-accessed embedding data for non-popular inputs. To perform the dynamic segregation and dispatching by Hotline, we offer a novel accelerator that can perform the fine grained tasks of determining which inputs require what embedding entries, and if needed gather them from main memory and dispatch the inputs. As such, Hotline hides the data transfer latency of obtaining non-frequently-accessed embedding data behind GPU execution of popular inputs.

We evaluate Hotline using four real world datasets using different recommender models. First, we found that Hotline outperforms the baseline for different types of models and on different datasets. Second, we observed that Hotline reduces the overall embedding transfer latency between CPU and GPU as popular inputs

do not need to obtain embeddings from main memory. Third, Hotline places the frequently-accessed embeddings closer to computational units (GPUs).
In conclusion, this thesis leverages frequently-accessed embeddings to speed up the training time for recommender models by reducing the communication latency and placing embeddings closer to computational unit without loosing accuracy.

8.2 Future Work

There are two potential areas in which this work can be extended.

8.2.1 Compressing Non-frequently-accessed Embeddings

As we have observed that frequently-accessed embeddings are just a small percentage of overall embedding size, but for recommender model to converge we need to train both frequently-accessed and non-frequently-accessed embeddings. But non-frequently-accessed embeddings can not fit within GPU memory because of size constraints. If we compress the non-frequently-accessed embeddings so that they fit within GPU memory then whole DLRM training can be shifted to GPU only training. Although this approach will have extra performance overhead of compression and decompression but keeping the access rate of non-frequently-accessed embeddings it would not be that much. One important aspect to be explored is the amount of compression that can be achieved within embedding weights across training epochs will be main motivation behind this future work.

8.2.2 Reducing Embedding Dimension and Embedding Precision

Another direction that can be explored would be embeddings dimension reduction along with embeddings precision. Embedding dimension is a hyper parameter which is fixed during training. If embedding dimension can be reduced in a dynamic way as per access rate of embeddings along-with their precision , then the overall communication latency can be reduced in the baseline execution. Prior work deals with this approach but with huge performance overhead.

Bibliography

[1] Mlperf becnhmarks https://mlcommons.org/en/training-normal-10/. ! page 36

[2] Nvlink. URL https://developer.nvidia.com/nccl. ! pages 8, 17

[3] Onnx runtime, 2019. URL https://github.com/microsoft/onnxruntime/blob/master/README.md. ! page 12

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensor ow: Large-scale machine learning on heterogeneous distributed systems, 2015. URL http://download.tensor ow.org/paper/whitepaper2015.pdf . ! pages 12, 35

[5] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood. Understanding training ef ciency of deep learning recommendation models at scale, 2020. ! pages 1, 11

[6] M. Adnan, Y. Ebrahimzadeh Maboud, D. Mahajan, and P. Nair. Accelerating recommendation system trainingby leveraging popular choices. PVLDB, 15(1): 127 - 140, 2022. doi:10.14778/3485450.3485462. URL https://doi.org/10.14778/3485450.3485462. ! pages v, 3

[7] Alibaba. User behavior data from taobao for recommendation. https://tianchi.aliyun.com/dataset/dataDetail?dataId=649&userId=1. ! pages 18, 28, 34

[8] F. Amato, V. Moscato, A. Picariello, and F. Piccialli. Sos: A multimedia recommender system for online social networks. Future Generation Computer Systems, 93:914 – 923, 2019. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2017.04.028. URL http://www.sciencedirect.com/science/article/pii/S0167739X17301693. ! page 10

[9] A. Anandhan, L. Shuib, M. A. Ismail, and G. Mujtaba. Social media recommender systems: Review and open research issues. IEEE Access, 6: 15608–15628, 2018. doi:10.1109/ACCESS.2018.2810062. ! page 10

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47, page 609–622, USA, 2014. IEEE Computer Society. ISBN 9781479969982. doi:10.1109/MICRO.2014.58. URL https://doi.org/10.1109/MICRO.2014.58. ! page 10

[11] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-ef cient data ow for convolutional neural networks. In Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16, page 367–379. IEEE Press, 2016. ISBN 9781467389471. doi:10.1109/ISCA.2016.40. URL https://doi.org/10.1109/ISCA.2016.40. ! pages 10, 12

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an ef cient and scalable deep learning training system. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 571–582, Broom eld, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi. ! pages 1, 8, 13

[13] K. Cho, M. Lee, K. Park, T. T. Kwon, Y. Choi, and Sangheon Pack. Wave: Popularity-based and collaborative in-network caching for content-oriented networks. In 2012 Proceedings IEEE INFOCOM Workshops, pages 316–321, 2012. doi:10.1109/INFCOMW.2012.6193512. ! page 3

[14] E. Chung, J. Fowers, K. Ovtcharov, , A. Caul eld, 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 12

[15] CriteoLabs. Criteo display ad challenge, . https://www.kaggle.com/c/criteo-display-ad-challenge. ! pages 18, 34

[16] CriteoLabs. Terabyte click logs, . https://labs.criteo.com/2013/12/download-terabyte-click-logs. ! pages 2, 18, 19, 34

[17] A. Desai, L. Chou, and A. Shrivastava. Random offset block embedding array (robe) for criteotb benchmark mlperf dlrm model : 1000 compression and 2.7faster inference. 2021. ! page 12

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018. ! page 1

[19] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti. Bandana: Using non-volatile memory for storing deep learning models. CoRR, abs/1811.05922, 2018. URL http://arxiv.org/abs/1811.05922. ! page 11

[20] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11, page 365–376, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304726. doi:10.1145/2000064.2000108. URL https://doi.org/10.1145/2000064.2000108. ! page 1

[21] 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 12

[22] B. Ghaemmaghami, Z. Deng, B. Cho, L. Orshansky, A. K. Singh, M. Erez, and M. Orshansky. Training with multi-layer embeddings for model reduction, 2020. ! pages 13, 14

[23] A. Ginart, M. Naumov, D. Mudigere, J. Yang, and J. Zou. Mixed dimension embeddings with application to memory-ef cient recommendation systems. ArXiv, abs/1909.11810, 2019. page 11

[24] C. A. Gomez-Uribe and N. Hunt. The net ix 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

[25] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. arXiv preprint arXiv:2001.02772, 2020. ! page 11

[26] 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. 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 488–501, 2020. doi:10.1109/HPCA47549.2020.00047. ! pages 11, 14

[27] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural collaborative ltering. 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 x, 8

[28] J.-T. Huang, A. Sharma, S. Sun, L. Xia, D. Zhang, P. Pronin, J. Padmanabhan, G. Ottaviano, and L. Yang. Embedding-Based Retrieval in Facebook Search, page 2553–2561. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379984. URL https://doi.org/10.1145/3394486.3403305. ! pages 1, 11

[29] 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: Ef cient 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-ef cient-training-of-giant-neural-networks-using-pipeline-parallelism . ! pages 1, 8, 10, 13

[30] R. Hwang, T. Kim, Y. Kwon, and M. Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. arXiv preprint arXiv:2005.05968, 2020. ! page 11

[31] 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 x, 1, 8, 11, 33

[32] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Ef cient data encoding for deep neural network training. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 776–789, 2018. doi:10.1109/ISCA.2018.00070. ! page 12

[33] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, page 60–71, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300537. doi:10.1145/1815961.1815971. URL https://doi.org/10.1145/1815961.1815971. ! page 26

[34] 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 1

[35] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, Proceedings of Machine Learning and Systems, volume 1, pages 1–13, 2019. URL https://proceedings.mlsys.org/paper/2019/ le/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf. ! page 13

[36] 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 5, 11, 14, 35

[37] 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 12

[38] Kaggle. Avazu mobile ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction. ! pages 18, 34

[39] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke. Optimizing deep learning recommender systems training on cpu cluster architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20. IEEE Press, 2020. ISBN 9781728199986. pages 5, 14, 35, 37

[40] 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. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 790–803, 2020. doi:10.1109/ISCA45697.2020.00070. ! pages 11, 14

[41] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy. Fbgemm: Enabling high-performance low-precision deep learning inference, 2021. ! page 14

[42] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30–37, 2009. doi:10.1109/MC.2009.263. ! page 11

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classi cation 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. ! pages 1, 7, 13

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classi cation with deep convolutional neural networks. Communications of the ACM, 60(6): 84–90, 2017. ! page 1

[45] H. Kwon, A. Samajdar, and T. Krishna. Maeri: Enabling exible data ow mapping over dnn accelerators via recon gurable interconnects. SIGPLAN Not., 53(2):461–475, Mar. 2018. ISSN 0362-1340. doi:10.1145/3296957.3173176. URL https://doi.org/10.1145/3296957.3173176. ! page 12

[46] Y. Kwon, Y. Lee, and M. Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 740–753, 2019. ! page 11

[47] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative ltering. IEEE Internet Computing, 7(1):76–80, 2003. doi:10.1109/MIC.2003.1167344. ! page 10

[48] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi:10.1145/2694344.2694358. URL http://doi.acm.org/10.1145/2694344.2694358. ! page 12

[49] H. Liu, X. Zhao, C. Wang, X. Liu, and J. Tang. Automated embedding size search in deep recommender systems. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in

Information Retrieval, SIGIR '20, page 2307–2316, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi:10.1145/3397271.3401436. URL https://doi.org/10.1145/3397271.3401436. ! pages 13, 14

[50] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal on Computing, 17(2): 373–386, 1988. ! page 28

[51] X. Luo, M. Zhou, Y. Xia, and Q. Zhu. An ef cient non-negative matrix-factorization-based approach to collaborative ltering for recommender systems. IEEE Transactions on Industrial Informatics, 10(2): 1273–1284, 2014. doi:10.1109/TII.2014.2308433. ! page 11

[52] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. Tabla: A uni ed template-based framework for accelerating statistical machine learning. 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 14–26, 2016. doi:10.1109/HPCA.2016.7446050. ! pages 11, 12

[53] G. E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8), April 1965. ! page 1

[54] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In MICRO, 2007. doi:http://dx.doi.org/10.1109/MICRO.2007.30. ! page 36

[55] 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 1, 8, 10, 13

[56] 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 x, 1, 8, 11, 33

[57] 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 11, 14

[58] Nvidia. NVIDIA Collective Communications Library (NCCL). https://docs.nvidia.com/deeplearning/nccl/index.html. ! page 35

[59] Nvidia. Accelerating wide deep recommender inference on gpus, 2017. https://developer.nvidia.com/blog/accelerating-wide-deep-recommender-inference-on-gpus/. ! page 11

[60] 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 12

[61] F. Papadopoulos, M. Kitsak, M. A. Serrano, M. Boguna, and D. Krioukov. Popularity versus similarity in growing networks. Nature, 489(7417): 537–40, Sep 27 2012. URL https://ezproxy.library.ubc.ca/login?url=https://www-proquest-com.ezproxy.library.ubc.ca/docview/1095114119?accountid=14656. Copyright - Copyright Nature Publishing Group Sep 27, 2012; Document feature - Illustrations; Graphs; ; Last updated - 2019-09-06; CODEN - NATUAS. page 3

[62] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh. Scale-out acceleration for machine learning. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17, page 367–381, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349529. doi:10.1145/3123939.3123979. URL https://doi.org/10.1145/3123939.3123979. ! pages 10, 11, 12

[63] 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. ! pages 12, 34

[64] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In 2016 ACM/IEEE 43rd

Annual International Symposium on Computer Architecture (ISCA), pages 267–278, June 2016. doi:10.1109/ISCA.2016.32. ! page 12

[65] C. Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft. Microsoft Blog 2019. ! pages 1, 13

[66] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen. Collaborative filtering recommender systems. In The adaptive web, pages 291–324. Springer, 2007. ! page 11

[67] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12, 2016. doi:10.1109/MICRO.2016.7783720. ! page 12

[68] 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. ! page 11

[69] 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 13

[70] 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

[71] 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 12

[72] 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. ! pages 8, 13

[73] G. Wang, D. Anand, N. Butt, A. Cestero, M. Chudzik, J. Ervin, S. Fang, G. Freeman, H. Ho, B. Khan, B. Kim, W. Kong, R. Krishnan, S. Krishnan, O. Kwon, J. Liu, K. McStay, E. Nelson, K. Nummy, P. Parries, J. Sim, R. Takalkar, A. Tessier, R. Todi, R. Malik, S. Stif er, and S. Iyer. Scaling deep trench based edram on soi to 32nm and beyond. In 2009 IEEE International Electron Devices Meeting (IEDM), pages 1–4, 2009. doi:10.1109/IEDM.2009.5424375. ! page 36

[74] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. R. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. ArXiv, abs/1910.04940, 2020. page 8

[75] Q. Wang, H. Yin, Z. Hu, D. Lian, H. Wang, and Z. Huang. Neural memory streaming recommender networks with adversarial training. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '18, page 2467–2475, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355520. doi:10.1145/3219819.3220004. URL https://doi.org/10.1145/3219819.3220004. ! page 10

[76] K. Wei, J. Huang, and S. Fu. A survey of e-commerce recommender systems. In 2007 International Conference on Service Systems and Service Management, pages 1–5, 2007. doi:10.1109/ICSSSM.2007.4280214. ! page 10

[77] C.-J. Wu, R. Burke, E. H. Chi, J. Konstan, J. McAuley, Y. Raimond, and H. Zhang. Developing a recommendation benchmark for mlperf training and inference, 2020. ! page 36

[78] 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-Paci c Workshop on Systems, pages 82–89, 2020. ! page 12

[79] C. Yin, B. Acun, X. Liu, and C.-J. Wu. Tt-rec: Tensor train compression for deep learning recommendation models, 2021. page 10

[80] S. Zhang, L. Yao, A. Sun, and Y. Tay. Deep learning based recommender system: A survey and new perspectives. ACM Comput. Surv., 52(1), Feb. 2019. ISSN 0360-0300. doi:10.1145/3285029. URL https://doi.org/10.1145/3285029. ! page 11

[81] 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. ! pages 1, 11, 14