









AtRec: Accelerating Recommendation Model Training on CPUs

Siqi Wang , Tianyu Feng , Hailong Yang , Xin You , Bangduo Chen , Tongxuan Liu ,
Zhongzhi Luan , and Depei Qian 

Abstract—The popularity of recommendation models and the enhanced AI processing capability of CPUs have provided massive performance opportunities to deliver satisfactory experiences to a large number of users. Unfortunately, existing recommendation model training methods fail to achieve high efficiency due to unique challenges such as dynamic shape and high parallelism. To address the above limitations, we comprehensively study the distinctive characteristics of recommendation models and discover several unexploited optimization opportunities. To exploit such opportunities, we propose *AtRec*, a high-performant recommendation model training engine that significantly accelerates the training process on CPUs. Specifically, *AtRec* presents comprehensive approach of training that employs operator-level and graph-level joint optimizations and runtime optimization. At the operator-level, *AtRec* identifies and optimizes the time-consuming operators, which enables further efficient graph-level optimizations. At the graph-level, *AtRec* conducts an in-depth analysis of the inefficiencies in several frequently used subgraphs, enables further performance improvement via eliminating redundant computations and memory accesses. In addition, to achieve better runtime performance, *AtRec* also identifies inefficiencies prevalent in the current scheduling and proposes runtime batching. The experiment results demonstrate that *AtRec* can significantly outperform state-of-the-art recommendation model training engines. We have open sourced the implementation and corresponding data of *AtRec* to boost research in this direction.

Index Terms—Recommendation system, deep neural network, model training, performance optimization.

I. INTRODUCTION

RECOMMENDATION systems are becoming an indispensable part of daily life and have been widely used in

Manuscript received 3 June 2023; revised 19 March 2024; accepted 19 March 2024. Date of publication 25 March 2024; date of current version 8 April 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB3001801, in part by the National Natural Science Foundation of China under Grant 62322201, Grant 62072018, Grant U23B2020, and Grant U22A2028, in part by the Fundamental Research Funds for the Central Universities under Grant YWF-23-L-1121, and in part by the State Key Laboratory of Software Development Environment under Grant SKLSDE-2023ZX-05. Recommended for acceptance by S. Wang. (*Siqi Wang and Tianyu Feng contributed equally to this work.*) (*Corresponding author: Hailong Yang.*)

Siqi Wang, Tianyu Feng, Hailong Yang, Xin You, Zhongzhi Luan, and Depei Qian are with the State Key Laboratory of Software Development Environment, and the School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: lethean1@buaa.edu.cn; ty_feng@buaa.edu.cn; hailong.yang@buaa.edu.cn; youxin2015@buaa.edu.cn; 07680@buaa.edu.cn; depei@buaa.edu.cn).

Bangduo Chen is with Alibaba Corporation, Beijing 100020, China (e-mail: chenbangduo.cbd@alibaba-inc.com).

Tongxuan Liu is with the University of Science and Technology of China, Hefei 230026, China (e-mail: tongxuan.ltx@mail.ustc.edu.cn).

AtRec is available at <https://github.com/buaa-hipo/Atrec>.
Digital Object Identifier 10.1109/TPDS.2024.3381186

e-commerce platforms [1], social media [2], video [3], music applications [4], and other fields to provide personalized and accurate recommendations for better user experience and business revenues. For example, Alibaba’s recommendation system has served over 800 million users worldwide, contributing to over 1 trillion dollars in Gross Merchandise Volume (GMV) [1]. YouTube’s recommendation system has helped over 1 billion users discover personalized content from an ever-growing library of videos [3], resulting in a 60% increase in clicks. Netflix’s recommendation system has generated a 75% increase in views and profits of over 1 billion dollars [5]. Spotify reported its significant monthly subscriber growth from 75 million to 100 million with the help of recommendation systems [4].

Although the concept of the recommendation system was introduced a long time ago [6], the success of industrial applications has continued to promote the rapid development of recommendation systems. It has gradually evolved from the initial collaborative filtering techniques to logistic regression models, and in recent years, deep learning-based recommendation models have been gaining significant attention. Specifically, DeepFM [7] and Wide & Deep [8] models have been widely deployed in the industry, effectively improving related applications. Moreover, DIEN [9] model is another widely adopted variant of the Wide & Deep model to incorporate time-series information such as historical user interests, bringing a new wave of development in recommendation systems.

When deploying deep learning-based recommendation systems in production, the performance of model training is of critical importance. For large-scale recommendation systems, the number of users and items is extremely large, and inefficient model training will increase the burden and cost of the recommendation system, delay the model updates, and further affect user experience. Therefore, the efficiency of the recommendation model training is of crucial importance to many enterprises. Meanwhile, CPU vendors continue to introduce new AI processing capabilities. For example, Intel Xeon Scalable Processors upgrades AVX256 to AVX512 and introduces an instruction set that supports BF16 data types for mixed-precision training, greatly improving the training performance. To explore the potential of CPU-based recommendation model training, we perform experiments comparing the performance of the training on CPU and GPU (detailed setup can be referred to Section IV-A). As shown in Fig. 1, in several cases, the *CPU-based* training achieves comparable performance compared to the *GPU-based* training. These experimental results indicate

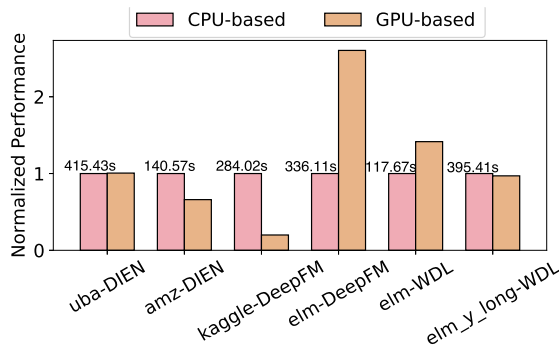


Fig. 1. Normalized performance with GPU-based recommendation model training (*GPU-based*) compared to CPU-based recommendation model training (*CPU-based*). The values on the bar of *CPU-based* represent its absolute latency incurred during the overall execution.

that the CPU-based and GPU-based recommendation model training have their respective advantages for different model and dataset. Especially, when considering the cost of up-front investment, CPU-based recommendation system is more appealing to medium and small organizations. Therefore, we believe that CPU-based optimizations are important in such scenarios and choose CPU as the target platform to optimize the performance of recommendation model training.

However, there is much room left for further improving the performance of recommendation model training on CPUs. The training of a recommendation model is essentially the execution of operators in its computation graph. Operator libraries such as OneDNN [10] and OpenBLAS [11] provide targeted optimization of the operator computation, that effectively accelerates the performance of the operators on specific platform. However, the implementation of these operator libraries requires significant tuning cost and their development usually lags behind the increasingly diverse model structures and operator representations. They mostly focus on general models such as Deep Neural Network (DNN) and Convolutional Neural Network (CNN) with computation-intensive operators (e.g., *Conv*, *MatMul*). Thus, such operator libraries will miss the opportunity to optimize specific operators for recommendation models (e.g., *StringSplitV2*, *OneHot*, etc.), which are time-consuming to process features during pre-processing. Besides, operator libraries are limited to the performance optimization of the individual operators, missing the opportunity for further graph-level optimizations.

Deep learning compilers serve as another means to accelerate model training [12], [13]. However, they can only achieve noticeable performance gains on general models such as DNN and CNN while failing to handle the unique challenges of recommendation models, resulting in limited performance. Specifically, the input of recommendation models is dynamic, while most of these compilers are designed to handle static-shape workloads. This implies that whenever there is a change in the input shape, recompilation becomes necessary, leading to significant compilation overhead. Besides, the recommendation model often contains many inexpensive operators with high parallelism, and the commonly adopted kernel fusion in deep learning compilers will miss parallelization opportunities, which

underutilizes the computation resources. The generality of deep learning compilers prevents them from performing specific optimizations tailored to the characteristics of recommendation models, resulting in suboptimal or even negative performance on recommendation models.

Based on the above analysis, we believe that to obtain better performance of recommendation model training on CPUs, it is necessary to adopt flexible and targeted optimizations according to the characteristics of recommendation models. To achieve the above goal, we propose *AtRec*, an efficient recommendation training engine based on DeepRec [14]. It optimizes the whole process of recommendation model training with several unique optimizations including operator-level and graph-level joint optimizations and runtime optimization. At the operator level, we identify the time-consuming operators and follow the well-studied techniques like vectorization and intra-operator parallelism to optimize them, which enables us for efficient graph-level optimization. At the graph level, we accelerate frequently used subgraphs via eliminating redundant computations and memory accesses. In addition, to improve runtime performance, we also identify inefficiencies prevalent in the current scheduling and propose runtime batching for best end-to-end performance. *AtRec* is open-sourced at <https://github.com/buaa-hipo/Atrec>. The experiment results demonstrate that *AtRec* can significantly outperform state-of-the-art recommendation model training engines.

Specifically, this paper makes the following contributions:

- We comprehensively analyze the performance bottlenecks of recommendation models and illustrate the opportunities to improve training throughput.
- We propose a series of operator-level and graph-level joint optimization schemes to achieve efficient recommendation model training. We also improve the runtime performance by employing runtime batching.
- We implement an efficient recommendation training engine *AtRec* and evaluate it with commonly adopted recommendation models. The evaluation results demonstrate that *AtRec* can achieve $1.07\times \sim 7.55\times$ speedup for end-to-end model training in comparison to the state-of-the-art training engines.

The rest of this paper is organized as follows. Section II introduces the backgrounds of deep learning-based recommendation models and key insights to motivate this work. Section III presents the details of the *AtRec* optimization strategies. We evaluate *AtRec* in Section IV. We discuss the related work in Section V and conclude this paper in Section VI.

II. BACKGROUND AND MOTIVATION

A. Deep Learning-Based Recommendation Frameworks

Fig. 2 shows the basic architecture of commonly adopted deep learning-based recommendation frameworks such as DIEN [9], WDL [8], DeepFM [7], etc., which typically consists of an embedding layer, a feature interaction layer, and a multilayer perceptron (MLP). The training of recommendation models mainly comprises two phases including *feature processing* and *feature interaction*. Specifically, for *feature processing*, the

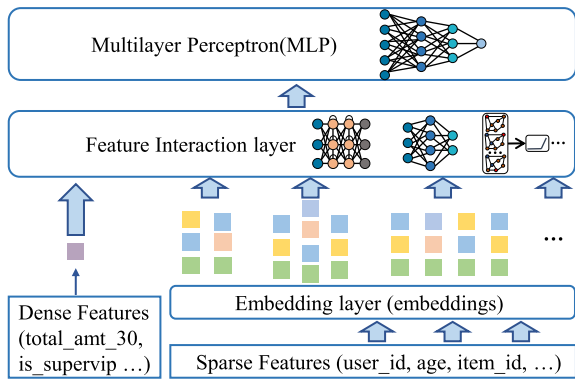


Fig. 2. Deep learning-based recommendation framework architecture.

sparse feature is first transformed into a low-dimensional dense feature by encoding various feature column data in the embedding layer and stored in the form of an embedding table. For *feature interaction*, recommendation frameworks will use a variety of different feature interaction modules (e.g., Recurrent Neural Network (RNN), Graph Neural Network (GNN), etc.) to extract useful information from embeddings or otherwise encoded inputs in the feature interaction layer, and their outputs will be concatenated as the output of this layer. Afterwards, the concatenated vector is fed into a fully-connected MLP to provide the final prediction.

Deep learning-based recommendation frameworks have been widely researched and applied in the industry. Wide & Deep (WDL) [8] is a recommendation framework proposed by Google, which skillfully combines traditional feature engineering with deep learning models. Specifically, it consists of *wide* and *deep* parts, where the *wide* part is a generalized linear model that performs memorization while the *deep* part is a feed-forward neural network that performs generalization. Starting from Wide & Deep, two trends have emerged in the innovation of industrial-scale recommendation models. First, the interaction between higher-order features and lower-order features is increased, so that feature representation can be performed more accurately. A typical framework of this kind is DeepFM [7], which improves on WDL by combining factorize machines (FM), in which the FM module and *deep* module share the same features to avoid feature engineering. Second, by introducing the attention mechanism, recommendation models can handle time-sequential features and derive the hidden user interests behind the historical behavior. A typical framework is DIEN [9], which incorporates the GRU (Gated Recurrent Unit) module to model the evolution of user interests.

B. Motivation

We make three key observations on the characteristics of recommendation models training on CPUs with new performance opportunities. The experimental setup can be referred to in Section IV-A with evaluated models shown in Table I.

Observation 1 (Expensive Feature Processing) - Commonly used datasets for recommendation models are always huge [15], [16], leading to feature processing as one of the bottlenecks when

TABLE I
MODELS USED FOR EVALUATION

Model	Dataset	Framework	Batch Size
amz-DIEN	amz_book [15]	DIEN	512
uba-DIEN	User Behaviour Dataset from Alibaba (UBA) [16]	DIEN	512
elm-DeepFM	elm [18]	DeepFM	4,096
kaggle-DeepFM	Kaggle Display Advertising Challenge dataset [19]	DeepFM	512
elm-WDL	elm [18]	WDL	4,096
elm_y_long-WDL	elm_y_long	WDL	4,096

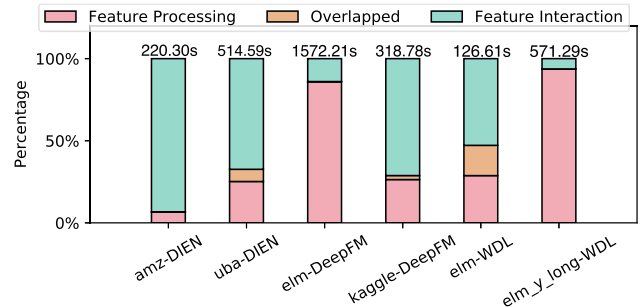


Fig. 3. Illustration of the training time breakdown of typical recommendation models. The time for each model is divided into three parts: feature processing, feature interaction, and the overlap between the two (overlapped). The values on each bar represent the absolute latency incurred during the overall execution.



Fig. 4. Partial timeline of *elm-WDL* performing feature processing.

training these models. Fig. 3 shows the breakdown of the training time of typical recommendation models when using the Deep-Rec engine, where the feature processing time is higher than 20% for most of the models. Specifically, the feature processing time of *elm_y_long-WDL* and *elm-DeepFM* become even higher for 93.73% (~535.47 seconds) and 85.95% (~1,351.31 seconds). Besides, *amz-DIEN* contains an RNN module, resulting in a longer feature interaction time. Although its data processing part only accounts for 6.55% of the full training process, it is still a significant overhead in absolute terms (~14.43 seconds). Figs. 4 and 5 show more detailed timelines of the feature processing process for *elm-WDL* and *elm-DeepFM*, respectively. The horizontal axis represents the execution time, whereas the

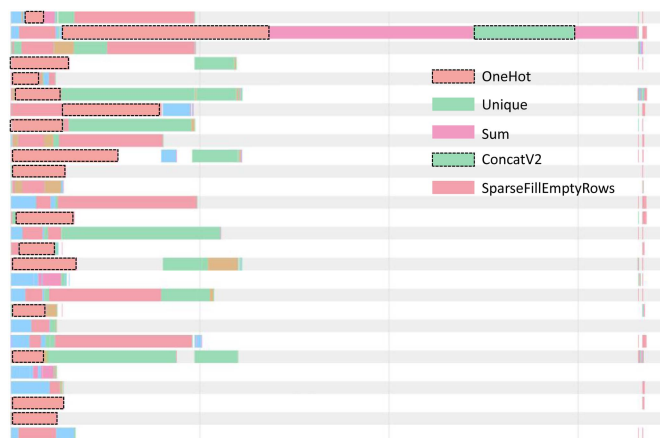


Fig. 5. Partial timeline of *elm-DeepFM* performing feature processing.

vertical axis depicts different execution threads. As shown in Fig. 4, in feature processing, the *StringSplitV2* operator is called frequently (light red color blocks with black border). This arises from the fact that the inputs of recommendation models contain a large amount of string-type sequential data, among which the historical features such as *shop_id_list* and *item_id_list* need to be split first. The splitting process will use *StringSplitV2* operators frequently, which generates non-negligible time consumption. Besides, as shown in Fig. 5, the *OneHot* operator is another hotspot operator in the feature processing (light red color blocks with black border). Specifically, *OneHot* operators are closely related to the feature column, which is an effective way to process word and ID features in the field of deep learning to map sample features into model-acceptable data for end-to-end training. In recommendation models, an indicator column is commonly used to convert the sparse tensor obtained from the categorical column into a dense tensor in the form of one-hot or multi-hot. As the indicator column adopts the encoding of one-hot, *OneHot* operators become a performance bottleneck in feature processing and even in the whole training process.

Observation 2 (Suboptimal Inexpensive Operator Scheduling Policy) - We observed a large number of inexpensive operators with concentrated execution and short running time in some of the recommendation models when using *DeepRec* engine. For example, in *kaggle-DeepFM*, the number of inexpensive operators is as high as 474 and the execution time accounts for 13.31%. Besides, by analyzing the timeline of each model, we found that most of these inexpensive operators are scheduled centrally in one thread with few dependencies, missing opportunities for concurrent scheduling. For instance, the feature processing leverages the *Variable* operator for storing the embeddings. The number of *Variable* operators will be considerably high if there are a large number of features to be embedded. However, *Variable* operators are independent of each other and the current centralized scheduling strategy does not fully utilize the CPU resources. The reason is that *DeepRec* is developed based on TensorFlow, without optimizations targeting recommendation models. For most DNN models, there are only a few inexpensive operators (typically consume a few microseconds), therefore TensorFlow believes it is profitable to execute these operators

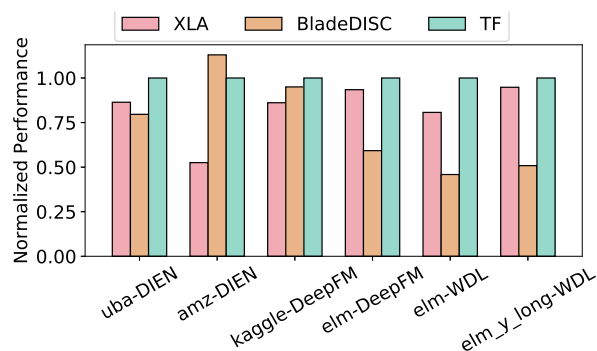


Fig. 6. Normalized performance with *XLA* and *BladeDISC* compared to *TF*.

within a single thread, other than executing them in parallel. This is because the overhead incurred by the additional scheduling outweighs the benefits gained from parallelism. However, for recommendation models, there are a significant number of inexpensive operators, therefore executing these operators in parallel becomes profitable for improving performance.

Observation 3 (Insufficient Pertinence of Compilers for Recommendation Models) - Existing deep-learning compilers cannot handle recommendation models to generate high-performant model implementations. Specifically, the optimization strategy of most compilers relies on static input and output shapes, and the dynamic shapes within the recommendation model inputs lead to additional overheads in the compilation process. Besides, the embedding layer and feature interaction layer of the recommendation model contains a large number of fragmented and highly parallelized operators, which are mostly fused to large operators by existing compilers, resulting in less inter-operator parallelism and under-utilization of multi-core CPU resources. As shown in Fig. 6, the *XLA* [12] compiler leads to even worse overall performance than native TensorFlow implementation [17] (a.k.a., *TF*). Specifically, *amz-DIEN* and *elm-WDL* on *XLA* result in $1.90\times$ and $1.24\times$ performance slowdown compared to *TF*, respectively. Even the best performant *elm_y_long-WDL* on *XLA* is still $1.05\times$ slower than *TF*. For *BladeDISC* [13], although it supports dynamic shapes, it also faces challenges posed by fragmented and highly parallelized operators of recommendation models. In addition, it only supports older versions of TensorFlow, leading to further performance degradation. Specifically, compared to *TF*, it exhibits a speedup of $1.13\times$ only on *amz-DIEN*, while performing poorly on other models, with even a $2.18\times$ slowdown on *elm-WDL*. Given the importance of recommendation models in business production, it is worthwhile to make the effort to optimize them manually.

In sum, the *observation 1* shows the hot spots in the whole process of recommendation model training and reveals an opportunity for operator optimizations within feature processing. The *observation 2* shows that there are a large number of inexpensive operations in the recommendation model training process which can be scheduled concurrently to further improve the training performance. The *observation 3* illustrates the incompatibility of existing deep learning compilers when dealing with recommendation models and raises the need for manual optimization.

III. METHODOLOGY

To mitigate the inefficiencies observed above and achieve outstanding performance on recommendation model training, we propose *AtRec*, a model training system tailored for vast kinds of contemporary recommendation models with a bunch of operator-level, graph-level, and runtime-level optimizations. Specifically, we systematically identify and conduct in-depth analyses of bottlenecks in *feature processing* and *feature interaction*. To improve training efficiency, we use operator-level and graph-level joint optimizations to eliminate redundant computations and memory accesses. To improve runtime performance, we leverage runtime batching to achieve efficient inexpensive operator scheduling. Moreover, we also perform additional auxiliary optimizations based on a few heuristic observations to further improve the end-to-end training efficiency. We implement the optimizations based on DeepRec, a recommendation engine derived from TensorFlow. A few optimizations for embedding, computation graph, and training runtime, which is orthogonal to our optimization approaches, have already been integrated into *AtRec*.

A. Feature Processing Optimization

1) *Operator-Level and Graph-Level Joint Optimization of StringSplit*: As mentioned in *observation 1*, historical sequential features are first split by *StringSplitV2* operators during the feature processing of the recommendation model, whose overhead is proportional to the length of the string to be split. Specifically, the built-in implementation of *StringSplitV2* in TensorFlow receives strings of arbitrary length as the delimiter splits the original string by repeatedly locating the next occurrence of the delimiter, and applies string slicing based on located indices. However, commonly used datasets in existing industrial recommendation models (e.g., Alibaba elm dataset [18]) use only single-character delimiters (e.g., semicolon). Unlike strings with variable lengths, the occurrences of a single character in a given string can be obtained using vectorized comparison operation, thus improving the throughput of the splitting process.

Therefore, we utilize this opportunity by implementing a vectorized version of *StringSplitV2* powered by AVX512. Specifically, as shown in Fig. 7, we load 64 bytes of the original string into a $64 \times \text{int8}$ 512-bit *zmm* register at a time (①), broadcast the delimiter character into another $64 \times \text{int8}$ 512-bit *zmm* register (②), and perform a vector comparison between two vector registers to obtain a 64-bit mask representing the separator's occurrences (③). Finally, the string-splitting process is completed by iterating through the delimiter indices in the mask using a series of basic operations composed of a CTZ (count trailing zeros) intrinsic and a logical bit shifting (④).

Moreover, there are subgraphs containing many operators with short execution times for sequence feature processing. As shown in Fig. 8, there are three frequently executed subgraphs during the training of the recommendation model, including *numerical data averaging subgraph*, *long sequence truncating subgraph*, and *categorical feature hash bucketing subgraph*.

Numerical data averaging subgraph is used to process numerical historical data such as *price_list*, *hours_list*, and so on.

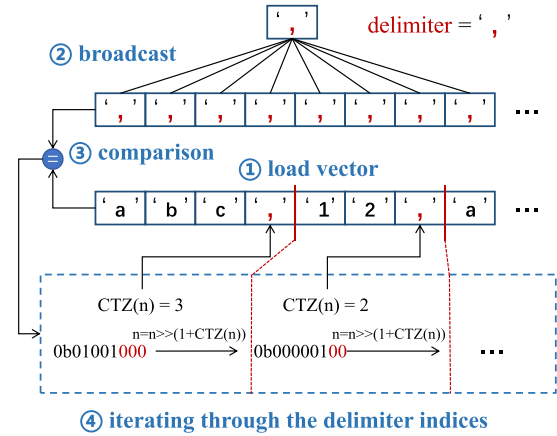


Fig. 7. Implementation process of the vectorized version of *StringSplitV2*.

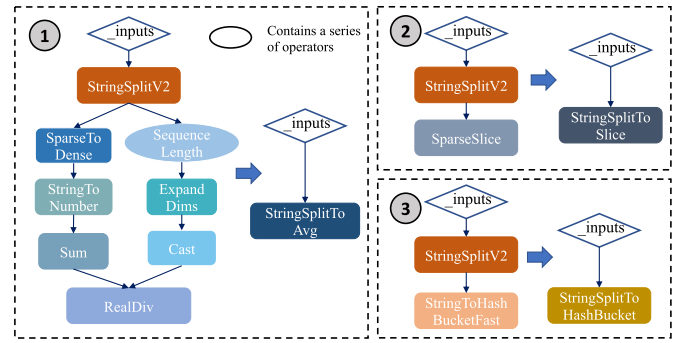


Fig. 8. Illustration of three inefficient subgraph patterns that frequently executed in the feature processing, including ① numerical data averaging subgraph, ② long sequence truncating subgraph, and ③ categorical feature hash bucketing subgraph.

Specifically, for substrings split by *StringSplitV2*, a *SparseToDense* operator converts the sparse features into a dense format, and then a *StringToNumber* operator converts the dense string features into tensors of a numerical data type. Afterward, a *Sum* operator is called to sum the features up. Meanwhile, a *Sequence Length* operator first obtains the length of substrings. Then an *ExpandDims* operator expands the dimension of the length tensor and a *Cast* operator converts its data type to float32. Finally, a *RealDiv* operator divides the sum by length to obtain the average value. As shown in Fig. 8①, the whole process is sequential while introducing multiple small operators, which causes unnecessary memory and scheduling overheads.

Long sequence truncating subgraph processes long string data such as historical item IDs and historical categorical features that exceed the maximum length configured for the sake of memory limitation or model scale. Specifically, as shown in Fig 8②, *StringSplitV2* is first leveraged to split the strings, and then *SparseSlice* is used to truncate the resulting tensor to the specified length limit. However, the truncation can also be achieved by early stopping *StringSplitV2* once after the specified number of tokens have been processed, eliminating redundant string splitting on truncated parts.

Categorical feature hash bucketing subgraph processes data containing a large number of categorical or numerical features

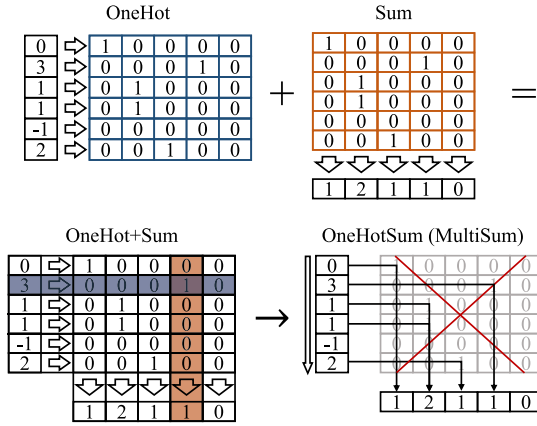


Fig. 9. Example of the *Onehot+Sum* operator fusion process.

in two phases as shown in Fig. 8③. Specifically, *StringSplitV2* first splits the strings into tokens, then these tokens are scattered into different corresponding hash buckets by their hash values (a.k.a., *StringToHashBucketFast*). However, the tokens can be directly hashed and assigned to hash buckets once it is split from the strings for better data locality, reducing the overhead of storing the intermediate results.

Based on the above insights, we apply fusion optimizations of sequence feature processing to replace the identified inefficient subgraphs above with high-performance fused operators, as shown in Fig. 8. *AtRec* searches the whole graph of feature processing for the above three subgraph patterns and applies operator fusion if possible.

2) *Onehot-Sum Redundant Computation Elimination*: The outputs of sequence feature processing are then fed to the embedding layer for further processes. Embedding layers are widely used in recommendation models to transform sparse high-dimensional features into dense low-dimensional features. One-hot encoding is one of the most common encoding methods within embedding layers, allowing different words to remain relatively independent while computing the correlation between them. However, although the *OneHot* operator has good scalability for parallelization, it involves a large number of computations to get a one-hot embedding vector with significant memory allocation. Besides, each *OneHot* operator is often followed by a *Sum* operator. Specifically, *OneHot* operates on the lowest dimension, and *Sum* computes on the penultimate dimension, which essentially counts the occurrences of different values among the lowest dimension of the original tensor (a.k.a. *MultiHot*). Fig. 9 illustrates an example to perform *OneHot+Sum* operation on a one-dimensional tensor (a.k.a., vector) with six elements. First, we use a *OneHot* operator to get the transformed one-hot embeddings, and then a *Sum* operator will sum the one-hot embeddings among the penultimate axis (along the vertical direction shown in Fig. 9) to get the final result. The whole process requires creating a 6×5 tensor storing the intermediate results of *OneHot*, which is prodigal in memory usage for implementing *MultiHot*. However, the two-stage process can be achieved by directly counting the number of occurrences of the elements in the original tensor. As shown in Fig. 9, the numbers

of occurrences of original element values 0, 1, 2, and 3 (which are 1, 2, 1, and 1, respectively) are accumulated and stored to the corresponding index of the resulting tensor as the element values, without generating intermediate results.

Therefore, we optimize the encoding of one-hot features with operator fusion. Specifically, we identify all the combinations of *OneHot+Sum* operators in the computation graph by constructing a graph fusion template and enforcing additional restrictions to filter out combinations that should not be handled. Then we replace all matched subgraphs with new *OneHotSum* (a.k.a. *MultiHot*) operators. A *OneHotSum* operator traverses the original tensor and counts the last dimension (i.e., the dimension on which the previously replaced *OneHot* operates) and records the result to the output tensor with the values of original tensor elements as indices. Note that the whole counting process can perform within the scope of a single sample and thus can be embarrassingly parallelized with different threads handling different samples in a batch since there is no data dependency between any of them. We use *shard* provided by Eigen to slice the task by samples and achieve intra-operator parallelism by utilizing the Eigen thread pool. Thus, even if the computations within a single data sample are serial, the whole process of computations is highly parallelized.

B. Feature Interaction Optimization

1) *GRU Cell Splitting*: Considering the changing external trends and internal aesthetic perceptions, users' interests also evolve dynamically over time. To predict users' interests more accurately, some recommendation models, including DIEN, have incorporated recurrent neural network (RNN) modules to extract temporal interest from users' historical behavior sequences. Gated recurrent unit (GRU) is widely used as it can effectively suppress vanishing gradient or exploding gradient compared to traditional RNNs, and its computational complexity is smaller than other RNN variants.

Due to the pervasiveness of GRU, *AtRec* implements highly optimized GRU for efficient training of such recommendation models. Specifically, as shown in the left part of Fig. 10, the *_inputs* and *state* required by reset (*r*) and update (*u*) gates are first concatenated. Then a *MatMul* operation is performed between the concatenated input and the weight of the same size (*Wru*) which represents the merger of the two gates. Afterward, a series of operations such as *Biasadd* and *Sigmoid* are performed on the result. Finally, the *Split* operation is used to divide the results into those corresponding to the reset gate (*r*) and the update gate (*u*). However, such *Concat-and-Split* approach is redundant and unnecessary. It incurs the overheads of redundant operators and limits parallelism in computation. We improve the computation process by separating the reset gate from the update gate. As shown on the right-hand side of Fig. 10, we no longer concatenate *_inputs* and *state*, and convert the previous large matrix multiplication into four small matrix multiplications of (*_inputs*, *Wr1*), (*state*, *Wr2*), (*_inputs*, *Wu1*), (*state*, *Wu2*). After the results are performed by *Biasadd*, *Add*, and *Sigmoid* operations, the final results can be obtained directly without splitting, saving unnecessary computation overhead.

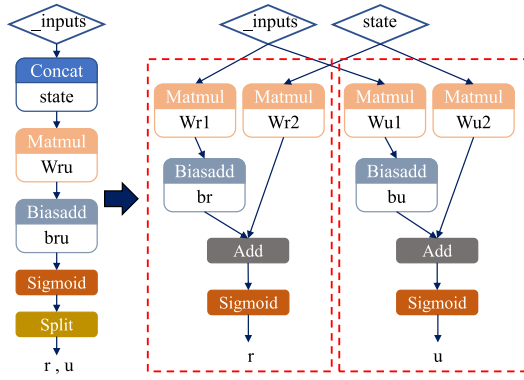


Fig. 10. Computation graph substitution of GRU cell splitting. The merged *MatMul* is replaced by four independent ones. Linear transformations on *_inputs* and *state* and computations of *u* and *r* gates are respectively decoupled from each other to avoid concatenation and splitting overhead. The new subgraph is equivalent to the substituted one from the perspective of computational semantics despite the order of operations is altered.

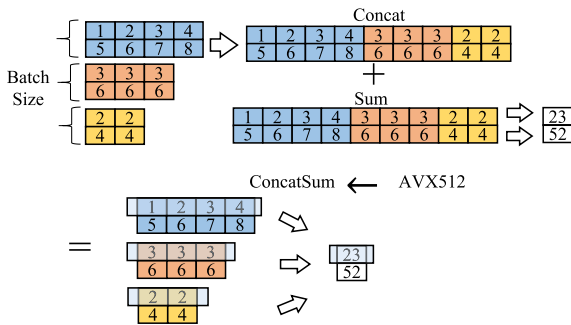


Fig. 11. Operator fusion of *Concat* and *Sum*. Vectorization is exerted on the kernel implementation to further accelerate the computation process.

2) *Feature Concatenation Optimization*: Besides, feature embeddings of the recommendation models will be concatenated before feeding to the MLP. If the feature embeddings generated by the model are too large, the *Concat* operator will become a performance bottleneck. For example, the *elm-DeepFM* model uses indicator columns when processing the feature, and its one-hot encoding results in large dense embeddings. Since the *Concat* operator makes a full copy of all input values (i.e., embeddings), the large input size would lead to non-negligible memory copy overhead. Meanwhile, the *Concat* operator in the *elm-DeepFM* model will be followed by a *Sum* operator which operates in the same dimension as the *Concat* operator. As shown in Fig. 11, the *Concat* operator first aggregates multiple tensors, after which the *Sum* operator sums along the dimension of the aggregation, which is equivalent to traversing multiple tensors and summing this dimension.

Therefore, we perform an operator fusion optimization to avoid redundant memory operations. Specifically, we identify eligible *Concat* and *Sum* operators in the model and implement a graph fusion template and corresponding fusion rules to combine each matched operator sequence into a single *ConcatSum* operator, thus saving a series of redundant memory allocation and copy overheads introduced by the *Concat* operator. As shown in the lower part of Fig. 11, the *ConcatSum* operator obtains

the final output by iterating over each input tensor, performing summation, and accumulating the summation results. The summation process of the fused operator is further vectorized to obtain higher throughput with AVX512 intrinsics.

C. Runtime Batching

TensorFlow is one of the industrial deep learning engines commonly used for recommendation models. When developing a scheduling policy, TensorFlow classifies all operators into expensive and inexpensive categories based on the computation overhead of the operator obtained by profiling. Its scheduling logic is based on the assumption that the scheduling overhead of dispatching an inexpensive operator to another thread is higher than that of executing the operator in the current thread. Therefore, during the scheduling process, the expensive operators are selected and assigned to other threads, while the inexpensive operators continue to wait for execution in the current thread. Although this reduces the scheduling overhead of inexpensive operators, this one-size-fits-all strategy may cause a large number of operators to be scheduled in one thread when the inexpensive operators are dense, blocking the execution of later operators on the critical path and leading to inefficient scheduling. As mentioned in *observation 2*, there are a large amount of independent inexpensive operators (e.g., *variable*, *identity*) in the recommendation model training.

Therefore, we implement runtime batching on inexpensive operators for more efficient operator scheduling. Specifically, we set two thresholds for the number of inexpensive operators, including Th_m and Th_b . For Th_m , we observe that the runtime batching is effective only when the number of inexpensive operators reaches a certain level such that their execution overhead is higher than the scheduling overhead. Therefore, we configure Th_m so that runtime batching will be used only if the number of inexpensive operators of the model reaches this threshold. For Th_b , we will count the number of inexpensive operators in the current queue during each ready node traversal. If this number exceeds Th_b , the *RunTask* function is triggered to launch another thread to directly schedule the operators currently in the queue. Large groups of consecutive inexpensive operators are thus dispatched evenly to multiple threads in parallel with less blocking time. Empirically, we set these two thresholds as 400 and 16, respectively. Fig. 12 shows a snippet of the timeline before and after runtime batching, where the intensive inexpensive operators before the optimization are successfully distributed to multiple threads, effectively improving the training performance.

D. OneDNN Rewriting Rules Adjustments

OneDNN is an open-source cross-platform performance library developed by Intel, which can effectively improve the performance of deep learning models on multiple platforms such as CPUs and GPUs. However, we find that some rewriting rules in OneDNN did not match the recommendation model, failing to achieve the expected performance results, and some even results in significant slowdowns. Specifically, we make the following observations: 1) the performance of OneDNN on small-size

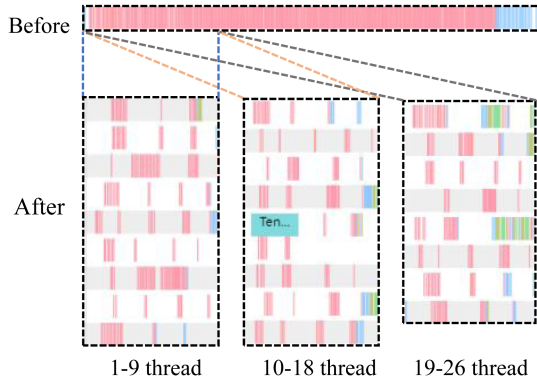


Fig. 12. Comparison between timeline tracings before and after runtime batching. The model used is *kaggle-DeepFM* with a batch size of 512. 26 of 28 threads in the inter-operator thread pool are utilized after this optimization, while only 1 thread is used instead without runtime batching.

MatMul is worse than the implementation in Eigen (the default math library of TensorFlow for CPU); 2) the performance of activation operators such as *Tanh* and *Sigmoid* provided in OneDNN is worse than that of Eigen, and its performance further deteriorates after being fused with the *MatMul* operator; 3) the *Reshape* will be converted into two *Const*, one *_MklReshape* and one *_MklToTf* after OneDNN’s optimization, resulting in a negative performance gain. Based on the above three observations, we make some adjustments to the remapper in the grappler and the *MklRewritingPass* in graph optimization, including (1) turning off the MKL rewriting optimization on small-size *MatMul*; (2) turning off the MKL rewriting optimizations of *Tanh*, *Sigmoid* and the fusion of *MatMul* with the two; (3) turning off the MKL rewriting optimization on *Reshape*.

IV. EVALUATION

A. Experimental Setup

1) *Hardware and Software Configurations*: We evaluate *AtRec* on a platform equipped with an Intel(R) Xeon(R) Gold 6330 CPU (28 physical cores running at 2.00 GHz, each containing two AVX512 FMA units) and 64 GB DDR4 memory. The experiments are conducted on Ubuntu 18.04 with GCC v10.3.0.

2) *Recommendation Models*: The evaluated recommendation models are presented in Table I. Specifically, we conduct a recommendation model by combining the training dataset and the recommendation framework. As shown in Table I, we utilize widely adopted click-through-rate (CTR) datasets as training datasets, including Amazon book review dataset *amz_book* [15], Kaggle Display Advertising Challenge dataset [19], User Behaviour Dataset from Alibaba [16], Alibaba cantering dataset *elm* [18] and a synthesized dataset *elm_y_long* where the length and proportion of historical sequences are increased based on *elm*. The evaluated recommendation frameworks are open-source frameworks widely used in academia and industry, including WDL, DIEN and DeepFM. The combination of datasets and frameworks constitutes the six models as shown in Table I.

3) *Comparison Methods*: We compare *AtRec* against TensorFlow (TF) [17], DeepRec [14], XLA [12], BladeDISC [13],

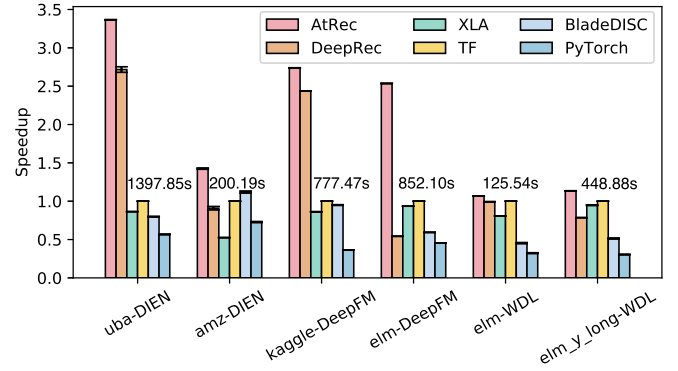


Fig. 13. End-to-end performance speedups of *AtRec*, DeepRec, XLA, BladeDISC, and PyTorch compared with TF on all evaluated models. The values on the bar of TF represent its absolute latency incurred during the overall execution. Each experiment is conducted three times and the average values are reported with error bars marked.

and PyTorch [20]. We choose the release version 2.11.0 for TF and XLA, and the release version 2.1.2 for PyTorch. Specifically, we conduct the evaluations to answer the following questions:

- 1) Compared with other engines, what is the overall performance of our proposed *AtRec*?
- 2) How much each proposed optimization strategy contributes to the overall performance?
- 3) How much impact do our proposed optimizations have on model accuracy?

B. End-to-End Performance

We first evaluate the overall training performance for the evaluated recommendation models compared with the baselines. For *AtRec*, the number of threads is configured to 28 for both inter- and intra-operator thread pools. The batch sizes used by each model are shown in Table I. All results are evaluated three times, with the average performance reported. Fig. 13 shows the overall performance speedups of end-to-end training with each evaluated method, where TF is demonstrated as the baseline. Overall, *AtRec* achieves better performance than all other benchmarks. Compared with XLA, TF, DeepRec, BladeDISC, and PyTorch, *AtRec* can achieve a speedup of $1.20\times$ – $3.89\times$, $1.07\times$ – $3.36\times$, $1.08\times$ – $4.68\times$, $1.26\times$ – $4.27\times$, and $1.94\times$ – $7.55\times$ respectively.

Specifically, as shown in Fig. 13, the generic deep learning frameworks TF and PyTorch are not optimal choices for the recommendation models, as they have no specific optimization and only optimize common operators in deep learning such as GEMM, convolution, etc. Although the XLA has performed a lot of compilation optimizations, it does not handle the characteristics of recommendation models, including input dynamics and high inter-operator parallelism. Therefore, XLA introduces additional overheads and misses some parallel opportunities, resulting in suboptimal performance result. BladeDISC only supports older versions of TensorFlow and performs inefficiently when encountering fragmented and highly parallelized operators of recommendation models, leading to inferior performance. DeepRec amalgamates various optimization techniques such

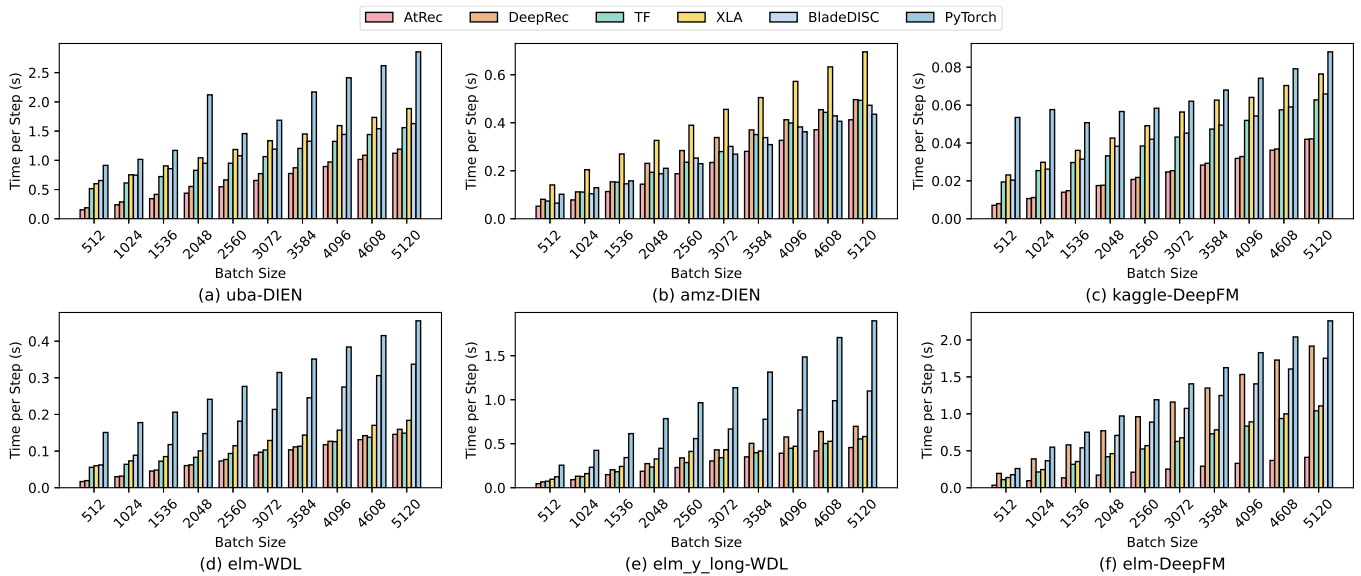


Fig. 14. Training time per step of six engines on six models in Table I with batch sizes ranging from 512 to 5,120.

as runtime optimization, operator optimization, and graph optimization for recommendation models to accelerate the training. *AtRec* further incorporates more intricate and effective optimizations for better performance. Compared with TF, XLA, DeepRec, BladeDISC, and PyTorch, *AtRec* can reach $2.04\times$, $2.50\times$, $1.85\times$, $2.87\times$, and $4.68\times$ average speedups, respectively.

To further validate the effectiveness of optimizations within *AtRec*, we evaluate the performance of each model with different batch sizes, as shown in Fig. 14. For *uba-DIEN* (Fig. 14(a)) and *amz-DIEN* (Fig. 14(b)), the optimization effect of *AtRec* is always significant as the batch size increases, outperforming XLA/TF/DeepRec/PyTorch/BladeDISC with the maximum speedup of $3.91\times/2.75\times/1.26\times/5.97\times/4.27\times$ and $2.70\times/1.41\times/1.60\times/1.95\times/1.35\times$, respectively, which proves the effectiveness of our optimization strategies.

For *kaggle-DeepFM*, the optimization is more effective when the batch size is small, and gradually decreases as the batch size increases, as shown in Fig. 14(c). The performance gain of this model mainly comes from the runtime batching of inexpensive operators. However, the time proportion in training gradually decreases with the increase of batch size, as the computation time of expensive operators increases significantly with the nearly unchanged cost of inexpensive operators. In practice, model developers rarely choose too large batch sizes due to longer training time [21]. When the batch size grows beyond a certain value, the training time of each epoch becomes shorter. However, the number of epochs required to reach the accuracy target increases at the same time, which leads to an increase in training time. Therefore, our optimization can still effectively accelerate the recommendation model training in practice (e.g., commonly adopted batch sizes listed in Table I). For *elm-DeepFM*, as shown in Fig. 14(f), *AtRec* results in a considerable optimization effect. Specifically, compared with XLA, TF, DeepRec, PyTorch, and BladeDISC, it gains a speedup of $2.59\times-4.03\times$, $2.25\times-3.20\times$,

$4.12\times-5.64\times$, $5.49\times-7.57\times$, and $3.84\times-5.13\times$ for all evaluated batch sizes.

For *elm-WDL* (Fig. 14(d)), *AtRec* still maintains notable speedups against the best-performant implementations, where our optimizations related to *StringSplitV2* operators contribute most of the performance improvements. However, the processing time of *StringSplitV2* is limited among the entire training process due to the restricted proportion of historical sequences to be split in the *elm* dataset. Moreover, with the increase of the batch size, the proportion of *StringSplitV2* reduces, resulting in a decreasing trend of speedup. Nevertheless, *AtRec* still achieves a speedup of $1.26\times-3.54\times$, $1.02\times-3.29\times$, $1.03\times-1.15\times$, $3.13\times-8.89\times$, and $2.31\times-3.67\times$ compared to XLA, TF, DeepRec, PyTorch, and BladeDISC respectively. For *elm_y_long-WDL* (Fig. 14(e)) with longer historical sequences, we observe that *AtRec* can gain a speedup of $1.19\times-2.11\times$, $1.12\times-1.64\times$, $1.37\times-1.53\times$, $3.73\times-5.63\times$, and $2.19\times-2.72\times$ compared to XLA, TF, DeepRec, PyTorch, and BladeDISC respectively. The percentage of *StringSplitV2* operators in the *elm_y_long* dataset increases for all batch sizes. Therefore, compared to DeepRec, *AtRec* obtains higher speedups at all batch sizes. For TF, although the maximum speedup is not improved, the decreasing trend of speedup is slower in *elm_y_long-WDL* as the batch size increases. Even at the maximum batch size, it can still achieve a speedup of $1.21\times$ (only $1.02\times$ in the *elm-WDL*), which illustrates the effectiveness of our optimizations. This is also true for XLA, but the fusion optimization for XLA is also more efficient when the percentage of *StringSplitV2* operators increases, so its slowing down of the decreasing trend is not as obvious as TF. Nevertheless, our optimization brings significantly better performance than its fusion optimization. We notice that PyTorch exhibits unusual poor performance on *uba-DIEN* with batch size of 2,048 and *kaggle-DeepFM* with batch size smaller than 1,024. The preliminary profiling

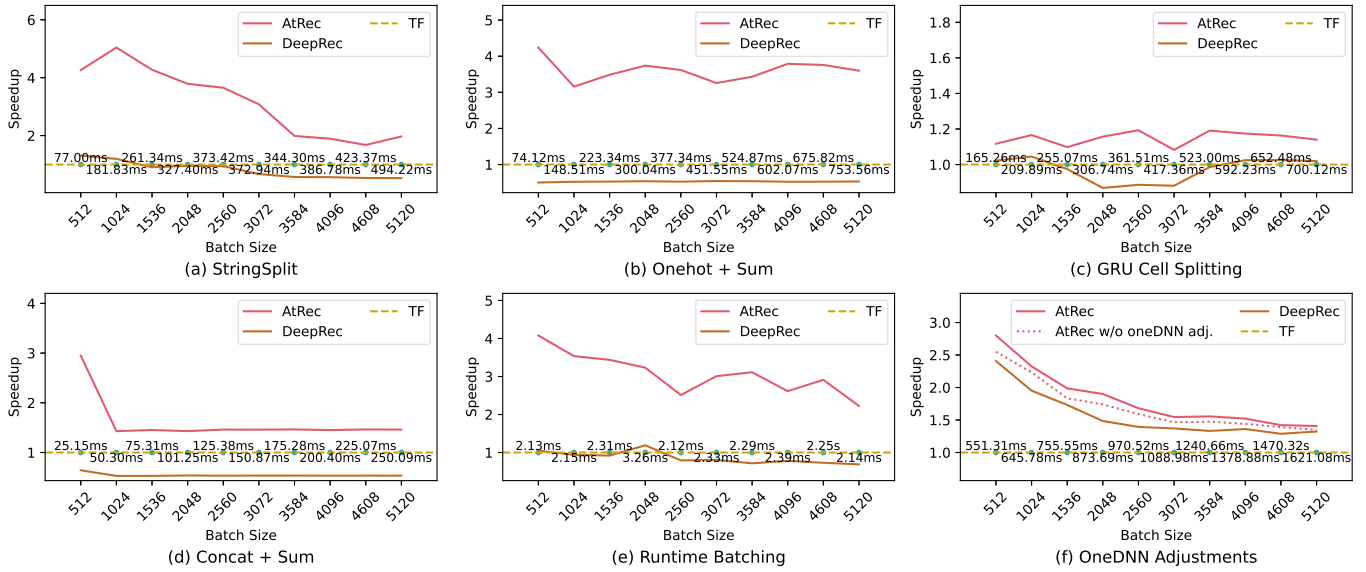


Fig. 15. Speedups gained by each optimization on different models, values on TF are absolute latencies per step—(a) speedup of cumulative duration of string-related operators by *operator-level and graph-level joint optimization of StringSplit* on *elm-WDL*; (b) speedup of combinations of *OneHot+Sum* operators by *OneHot-Sum redundant computation elimination* on *elm-DeepFM*; (c) speedup of RNN loops by *GRU cell splitting* on *uba-DIEN*; (d) speedup of *Concat + Sum* operators by *feature concatenation optimization* on *elm-DeepFM*; (e) speedup of the execution of inexpensive operators by *runtime batching* on *kaggle-DeepFM*; (f) speedup of the end-to-end training time by *OneDNN rewriting rules adjustments* on *uba-DIEN*.

results show that the poor performance can be attributed to the unexpected long backward propagation process. However, a further investigation is impeded by the opaqueness of PyTorch’s autograd engine.

C. Optimization Implications

To demonstrate the benefits of our optimizations in detail, we further break down the end-to-end performance gains of our optimization strategies and investigate them individually. For each optimization, we select 1–2 models from the corresponding network family with similar characteristics. Other models in the same family not listed are able to gain similar performance improvements. It is worth noting that XLA is excluded from the system to be compared. One reason is that XLA always performs worse than DeepRec or TF in our evaluation, making it reasonable to compare only with DeepRec and TF to verify the effectiveness of our optimization. Another is that XLA fuses most of the computation subgraphs into monolithic operators, making it difficult to analyze the performance of the computation process we are interested in.

1) *Operator-Level and Graph-Level Joint Optimization of StringSplit*: Fig. 15(a) shows the performance speedups of only applying these optimizations. As shown in Fig. 15(a), the effectiveness is affected by the batch size with speedups of $1.68\times$ – $5.04\times$. Specifically, the effectiveness of the optimization is closely related to the length and proportion of historical sequences in the dataset used by the model. Generally, the more and longer historical sequences in the training dataset result in more significant speedups. For illustration, we lengthened the historical sequences including *shop_id_list* and *item_id_list* in the *elm* dataset and increased their proportion to obtain the synthesized *elm_y_long* dataset. Specifically, the *elm_y_long* dataset has $8\times$

longer historical sequences with all hashes truncated to eight characters to offset the growth in file size. For *elm_y_long-WDL*, we observe the significant increased speedups of 3.39 – $7.46\times$ as expected. Even though we have only demonstrated the results on two models, it is notable that our optimizations are capable of achieving comparable optimization results on other models that have a fair amount of historical sequences as well.

2) *Onehot-Sum Redundant Computation Elimination*: Fig. 15(b) illustrates the performance of *OneHot+Sum* with only applying onehot-sum redundant computation elimination (a.k.a., *OneHotSum*). Specifically, *AtRec* achieves a speedup of about $3.5\times$ despite the changes in batch size. *DeepRec* is significantly slower than *AtRec* and TF due to different reasons. Specifically, TF outperforms *DeepRec* because of the more efficient *Onehot* operator implementation, while *AtRec* eliminates redundant computations to achieve better performance than TF. For *elm-DeepFM* configured as Table I, the optimization results in an overall end-to-end training speedup of $3.90\times$. Generally, the optimization can achieve a similar optimization effect in the model using indicator columns for feature processing.

3) *GRU Cell Splitting*: Fig. 15(c) shows the performance of GRU cell splitting for the GRU forward and backward propagation process. By comparing the timeline traces generated before and after splitting, it can be seen that splitting makes part of subgraphs starting from *input* in the GRU cell independent of the previous states so that their dependencies are stripped from the loop. This part of the subgraph includes *MatMul* and *BiasAdd* which only involve inputs, and their results can be computed at the beginning of the GRU loop without waiting for the loop to iterate to the corresponding iteration, thus having a high degree of parallelism and accelerating the loop execution.

This optimization accelerates the entire training of *uba-DIEN* and *amz-DIEN* with overall speedups of $1.14\times$ and $1.44\times$, respectively. Such optimization techniques are not limited to GRU but work on all recommendation models involving RNN with multiple gates in the recursive cell (e.g., LSTM).

4) *Feature Concatenation Optimization*: Fig. 15(d) shows the performance of *Concat+Sum* with only applying feature concatenation optimization (a.k.a., *ConcatSum*). The speedups of *AtRec* decrease significantly when the batch size increases to 1,024. The reason is that as the batch size increases, the memory consumption of operators exceeds the cache capacity and thus incurs additional cache miss overhead. However, our optimizations still remain effective by achieving $1.45\times$ performance improvement, which can be attributed to the reduced memory allocation and copy overhead during concatenation. For the entire training of *elm-DeepFM*, the optimization achieves an overall speedup of $1.18\times$. It is worth noting that the optimization effect can be seen in all models that require concatenating processed features, and rises as the number of features increases.

5) *Runtime Batching*: Fig. 15(e) shows the performance results of inexpensive operators with different batch sizes. It can be seen that runtime batching achieves a considerable optimization effect due to dispatching a large number of aggregated inexpensive operators to multiple threads. Larger batch sizes have negative influences on the effectiveness optimization and result in the descending curve shown in Fig. 15(e). Nevertheless, *AtRec* still results in notable speedups when scheduling inexpensive operators even with a large batch size of 5,120. Specifically, *AtRec* achieves up to $4.08\times$ speedup with a batch size of 512. For *kaggle-DeepFM*, the optimization achieves an overall speedup of $1.11\times$. Generally, the optimization achieves similar results in other models with a large number of inexpensive operators typically introduced by frequent embedding variable fetching.

6) *OneDNN Rewriting Rules Adjustments*: We performed an end-to-end performance comparison between *AtRec*, *AtRec w/o OneDNN adj.* (*AtRec* without OneDNN rule adjustments), and other engines. The results are shown in Fig. 15(f). In our evaluations, the OneDNN rule adjustments deliver notable performance optimizations for small-size matrix multiplication, which is common in the recurrent cell of the RNN recommendation model (e.g., DIEN). Taking *uba-DIEN* as an example, the OneDNN rule adjustments further achieve a maximum speedup of $1.10\times$ for *AtRec*. The optimization effect is influenced by the batch size, with larger batch sizes resulting in increased matrix multiplication size and decreased speedups. Generally, these adjustments can be applied selectively by automatically choosing the implementations of the best performance based on the parameters and type of operators, and we leave this as future work.

In sum, *AtRec* proposes optimizations from multiple perspectives and scales at the operator level, graph level, and system level for the complete process from feature processing to feature interaction, covering a variety of recommendation models. It achieves considerable speedup ratios on optimization targets and has significant improvements in the end-to-end performance of overall model training. The evaluated models are highly representative ones as most of today's prevailing recommendation

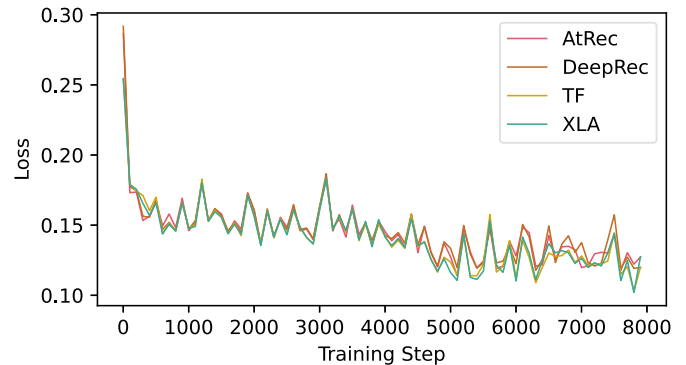


Fig. 16. Loss curves of *AtRec*, *DeepRec*, *TF*, and *XLA* during the process of training *elm-DeepFM* with a batch size of 4,096.

TABLE II
AUC OF TRAINED MODELS

Model	AUC			
	<i>AtRec</i>	<i>DeepRec</i>	<i>TF</i>	<i>XLA</i>
amz-DIEN	0.74	0.74	0.74	0.74
uba-DIEN	0.83	0.83	0.84	0.84
elm-DeepFM	0.56	0.57	0.57	0.56
kaggle-DeepFM	0.79	0.79	0.78	0.78
elm-WDL	0.57	0.57	0.56	0.56
elm_y_long-WDL	0.56	0.56	0.55	0.55

models are derived from or related to the evaluated models. For those models that are not evaluated due to the huge diversity of recommendation models, we believe that the proposed optimizations in *AtRec* are still beneficial and applicable, making *AtRec* expected to effectively accelerate the model training and reach a comparable performance with or even outperform existing model training engines including *DeepRec* and *TF*.

D. Accuracy

All the designs and optimizations in *AtRec* preserve the original semantics of models and should have a minor impact on the model training process and accuracy results. Fig. 16 shows the loss curves training with *AtRec*, *DeepRec*, *TF*, and *XLA* with identical hyper-parameters (e.g., batch size) on the same model. Specifically, different engines have almost the same trend of loss variation. The consistency of losses is further proved by that the maximum difference in losses between *AtRec* and the baseline engines (0.032) does not exceed the maximum variance observed between any two of them (0.037). The variance exists due to the stochastic nature of the recommendation model training itself.

We utilize AUC, the widely adopted CTR metric to evaluate the accuracy of trained recommendation models [7], [22]. AUCs of models trained with *AtRec*, *DeepRec*, *TF*, and *XLA* are shown in Table II. After training with the same batch size for the same number of iterations, *AtRec* obtains similar or higher AUCs compared with *XLA*, *TF*, and *DeepRec*. Even in the worst case, the difference between the AUCs of *AtRec* and these three benchmarks does not exceed 0.01. Although the results keep relatively consistent among different engines, they are not identical. The reasons for this include floating point errors

caused by altered computation order and the randomness introduced by model parameter initialization with unfixed random seeds. Such difference is consistent with the stochastic nature of model training and thus acceptable. This experiment proves that *AtRec* can achieve higher performance without sacrificing the convergence and accuracy of the trained model.

V. RELATED WORK

Deep Learning Operator Library - Deep learning operator libraries are widely used by researchers due to their efficiency and efficacy [10], [11], [23], [24], [25], [26], [27], [28], [29]. OneDNN [10] is Intel's open-source cross-platform deep learning performance acceleration library which provides highly optimized implementations of deep learning building blocks. By using OneDNN, users can accelerate programs on specific platforms without writing any target-specific code. OpenBLAS [11] is an optimized BLAS (Basic Linear Algebra Subprograms) library with many hand-crafted optimizations for specific processors. NNPACK [23] is a package for accelerated neural network computing, which can provide high-performance convolution layer implementations for multi-core CPUs. However, the development of operator libraries is often arduous and cumbersome, requiring a lot of manpower and leading to heavy development costs. Therefore, these libraries generally only include common compute-intensive operators (e.g., *Conv*, *MatMul*) and memory-intensive operators (e.g., *BatchNorm*, *Pooling*), and there is almost no specific optimization for operators which are much more prevalent in recommendation models (e.g., *StringSplit*, *OneHot*).

Deep Learning Compilers - There have been many efforts to build deep learning compilers with efficiency and portability [12], [30], [31], [32], [33], [34], [35], [36], [37]. XLA [12] integrates multiple graph optimization and parameter optimization methods and generates fused computation graphs by compiling the TensorFlow computation graph. AStitch [30] proposes a compilation optimization method for coarse-grained computing integration, which automatically generates efficient code through the joint consideration of dependency characteristics, memory hierarchy, and parallelism. TensorIR [31] proposes a new abstraction for tensor programs, which separates tensor computations from loop transformations and generates high-performance code using automated scheduling algorithms and other optimizations. However, on one hand, they are seldom able to support model training, while on the other hand, they cannot effectively handle the dynamic shapes and high concurrency of the recommendation model. Furthermore, the deep learning compiler does no targeted optimizations for characteristic procedures and operations in recommendation models such as feature processing and embedding, and there is still room for improvement in performance.

Recommendation Model Training Optimizations - Researchers have proposed various innovations to improve the performance of recommendation model training [22], [38], [39], [40], [41], [42], [43], [44], [45], [46]. Meta [38] points out the importance of the DSI pipeline in large-scale recommendation

model training and presents data storage, ingestion, and a training pipeline for production recommendation models. Neo [40] proposes a software-hardware collaborative system designed for DLRMs training, using various optimization methods including 4D parallelism, hybrid kernel fusion, and software-managed caching to achieve an efficient training process. AutoShard [41] optimizes embedding table sharding, leverages a neural cost model to efficiently predict the table cost, and uses deep RL to solve the partition problem. These works focus more on training optimization for distributed systems. Our work is orthogonal to them, and the overall performance can be further improved by optimizing the model training process on every single node.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a recommendation training engine *AtRec*, which can efficiently train recommendation models on CPUs. *AtRec* accelerates model training through operator-level and graph-level joint optimizations and runtime optimization. At the operator-level, *AtRec* identifies time-consuming operators and optimizes them, which enables further efficient graph-level optimizations. At the graph-level, *AtRec* accelerates several inefficient subgraphs via eliminating redundant computations and memory accesses. Moreover, *AtRec* also employs runtime batching to improve runtime performance. The experiment results demonstrate that *AtRec* can achieve average performance speedups of $2.04\times$, $2.50\times$, $1.85\times$, $2.87\times$, and $4.68\times$ compared to TF, XLA, DeepRec, BladeDISC, and PyTorch respectively.

For future work, we would like to adapt our method to GPU-based recommendation model training. Based on our preliminary studies, we have observed that the bottlenecks identified during CPU-based training persist in GPU-based training. Therefore, we believe our proposed optimizations remain useful in GPU-based training. Specifically, for operator-level optimizations, they can be still effective by adapting to the hardware difference between CPU and GPU. Whereas for graph-level optimizations and runtime batching, they are independent from the hardware architecture, and thus can be easily applied to GPU-based training.

REFERENCES

- [1] Y. Gu et al., "Self-supervised learning on users' spontaneous behaviors for multi-scenario ranking in e-commerce," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage.*, 2021, pp. 3828–3837.
- [2] Y.-F. Huang and P.-L. Wang, "Picture recommendation system built on Instagram," in *Proc. Int. Conf. Artif. Intell. Automat. Control Technol.*, 2017, pp. 1–6.
- [3] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for YouTube recommendations," in *Proc. 10th ACM Conf. Recommender Syst.*, 2016, pp. 191–198.
- [4] C. Underwood, "Use cases of recommendation systems in business-current applications and methods," *May*, vol. 20, 2019, Art. no. 2019.
- [5] C. A. Gomez-Urbe and N. Hunt, "The Netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, pp. 1–19, 2015.
- [6] J. Karlgren, "An algebra for recommendations: Using reader data as a basis for measuring document proximity," Dept. Comput. Syst. Sci., Stockholm Univ., 1990.
- [7] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "DeepFM: A factorization-machine based neural network for CTR prediction," 2017, *arXiv:1703.04247*.

- [8] H.-T. Cheng et al., "Wide & deep learning for recommender systems," in *Proc. 1st Workshop Deep Learn. Recommender Syst.*, 2016, pp. 7–10.
- [9] G. Zhou et al., "Deep interest evolution network for click-through rate prediction," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 5941–5948.
- [10] Intel oneDNN, 2016. [Online]. Available: <https://github.com/oneapi-src/onednn>
- [11] OpenBLAS, 2013. [Online]. Available: <https://www.openblas.net/>
- [12] TensorFlow XLA, 2021. [Online]. Available: <https://www.tensorflow.org/xla>
- [13] Z. Zheng et al., "BladeDISC: Optimizing dynamic shape machine learning workloads via compiler approach," in *Proc. ACM Manage. Data*, vol. 1, no. 3, pp. 1–29, 2023.
- [14] Alibaba, "DeepRec," 2022. [Online]. Available: <https://deeprec.readthedocs.io/zh/latest/>
- [15] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes," in *Proc. 38th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2015, pp. 43–52.
- [16] H. Zhu et al., "Learning tree-based deep model for recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 1079–1088.
- [17] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
- [18] Alibaba, "DeepRec CTR model performance optimization competition," 2022. [Online]. Available: <https://tianchi.aliyun.com/dataset/145340>
- [19] Kaggle, "Kaggle display advertising challenge dataset," 2014. [Online]. Available: <http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/>
- [20] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 721.
- [21] G. Zhang et al., "Which algorithmic choices matter at which batch sizes? Insights from a noisy quadratic model," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8194–8205.
- [22] Y. Zhang et al., "PICASSO: Unleashing the potential of GPU-centric training for wide-and-deep recommender systems," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 3453–3466.
- [23] NNPACK, 2016. [Online]. Available: <https://github.com/maratyszcza/nnpack>
- [24] J. Bi et al., "Heron: Automatically constrained high-performance library generation for deep learning accelerators," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 314–328.
- [25] XNNPACK, 2019. [Online]. Available: <https://github.com/google/xnnpack>
- [26] M. Tanvir, K. Narasimhan, M. Goli, O. El Farouki, S. Georgiev, and I. Ault, "Towards performance portability of ai models using SYCL-DNN," in *Proc. Int. Workshop OpenCL*, 2022, pp. 1–3.
- [27] J. Meng et al., "Automatic generation of high-performance convolution kernels on ARM CPUs for deep learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2885–2899, Nov. 2022.
- [28] H. Lan et al., "FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 580–594, Mar. 2020.
- [29] Arm ArmNN, 2018. [Online]. Available: <https://developer.arm.com/tools%20and%20software/armnn>
- [30] Z. Zheng et al., "AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 359–373.
- [31] S. Feng et al., "TensorIR: An abstraction for automatic tensorized program optimization," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 804–817.
- [32] H. Zhu et al., "ROLLER: Fast and efficient tensor compilation for deep learning," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 233–248.
- [33] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 47–62.
- [34] K. Zhu et al., "DISC: A dynamic shape compiler for machine learning workloads," in *Proc. 1st Workshop Mach. Learn. Syst.*, 2021, pp. 89–95.
- [35] R. Baghdadi et al., "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2019, pp. 193–205.
- [36] H. Wang et al., "PET: Optimizing tensor programs with partially equivalent transformations and automated corrections," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 37–54.
- [37] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," 2018, *arXiv: 1802.04799*.
- [38] M. Zhao et al., "Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 1042–1057.
- [39] Y. Huang et al., "Hierarchical training: Scaling deep recommendation models on large CPU clusters," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2021, pp. 3050–3058.
- [40] D. Mudigere et al., "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 993–1011.
- [41] D. Zha et al., "AutoShard: Automated embedding table sharding for recommender systems," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2022, pp. 4461–4471.
- [42] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "RecShard: Statistical feature-based memory optimization for industry-scale neural recommendation," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 344–358.
- [43] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, "EL-Rec: Efficient large-scale recommendation model training via tensor-train embedding table," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1007–1020.
- [44] X. Lian et al., "Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2022, pp. 3288–3298.
- [45] M. Xie et al., "Kraken: Memory-efficient continual learning for large-scale real-time recommendations," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–17.
- [46] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryayev, and A. Heinecke, "Optimizing deep learning recommender systems training on CPU cluster architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–15.



Siqi Wang is currently working toward the Ph.D. degree with the School of Computer Science and Engineering, Beihang University. She is currently working on domain-specific compiler and deep learning performance optimization. Her research interests include HPC and deep learning.



Tianyu Feng is currently working toward the M.S. degree with the School of Computer Science and Engineering, Beihang University. He is currently working on performance analysis, deep learning performance optimization, and power efficiency. His research interests include HPC, deep learning systems, and sustainable computing.



Hailong Yang received the Ph.D. degree from the School of Computer Science and Engineering, Beihang University, in 2014. He is a professor with the School of Computer Science and Engineering, Beihang University. He has been involved in several scientific projects such as performance analysis for Big Data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization, and energy efficiency.



Xin You is currently working toward the Ph.D. degree with the School of Computer Science and Engineering, Beihang University. He is currently working on performance analysis and performance optimization. His research interests include performance analysis, HPC, and deep learning.



Zhongzhi Luan received the Ph.D. degree from the School of Computer Science, Xi'an Jiaotong University. He is an associate professor of computer science and engineering, and assistant director with the Sino-German Joint Software Institute (JSI) Laboratory, Beihang University, China. Since 2003, his research interests include distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.



Bangduo Chen received the master's degree from Beihang University, in 2021. He is currently working with Alibaba, and his research interests include developing and optimizing the machine learning engine.



Depei Qian received the master's degree from the University of North Texas, in 1984. He is a professor with the School of Computer Science and Engineering, Beihang University, China. He is also a fellow with China Computer Federation (CCF). His research interests include innovative technologies in distributed computing, high performance computing, and computer architecture.



Tongxuan Liu is currently working toward the Ph.D. degree with the School of Computer Science and Technology, University of Science and Technology of China. He was the leader of the DeepRec Team, Alibaba, and in charge of DeepRec research and development, engaged in R&D of machine learning platforms.