RECom: A Compiler Approach to Accelerate Recommendation Model Inference with Massive Embedding Columns

Zaifeng Pan*
Renmin University of China
Beijing, China
panzaifeng@ruc.edu.cn

Zhen Zheng
Alibaba Group
Hangzhou, China
james.zz@alibaba-inc.com

Feng Zhang
Renmin University of China
Beijing, China
fengzhang@ruc.edu.cn

Ruofan Wu*
Renmin University of China
Beijing, China
ruofanwu@ruc.edu.cn

Hao Liang
Alibaba Group
Hangzhou, China
yunjing.lh@alibaba-inc.com

Dalin Wang*
Renmin University of China
Beijing, China
sxwangdalin@ruc.edu.cn

Xiafei Qiu
Alibaba Group
Hangzhou, China
xiafei.qiuxf@alibaba-inc.com

Junjie Bai
Alibaba Group
Hangzhou, China
j.bai@alibaba-inc.com

Wei Lin
Alibaba Group
Hangzhou, China
weilin.lw@alibaba-inc.com

Xiaoyong Du
Renmin University of China
Beijing, China
duyong@ruc.edu.cn

ABSTRACT

Embedding columns are important for deep recommendation models to achieve high accuracy, but they can be very time-consuming during inference. Machine learning (ML) compilers are used broadly in real businesses to optimize ML models automatically. Unfortunately, no existing work uses compilers to automatically accelerate the heavy embedding column computations during recommendation model inferences. To fill this gap, we propose RECom, the first ML compiler that aims at optimizing the massive embedding columns in recommendation models on the GPU. RECom addresses three major challenges. First, generating an efficient schedule on the GPU for the massive operators within embedding columns is difficult. Existing solutions usually lead to numerous small kernels and also lack inter-subgraph parallelism. We adopt a novel codegen strategy that fuses massive embedding columns into a single kernel and maps each column into a separate thread block on the GPU. Second, the complex shape computations under dynamic shape scenarios impede further graph optimizations. We develop a symbolic expression-based module to reconstruct all shape computations. Third, ML frameworks inevitably introduce redundant computations due to robustness considerations. We develop a subgraph optimization module that performs graph-level simplifications based on the entire embedding column context. Experiments on both in-house and open-source models show that RECom can achieve 6.61× and 1.91× over state-of-the-art baselines in terms of end-to-end inference latency and throughput, respectively. RECom's source code is publicly available at https://github.com/AlibabaResearch/recom.

*Work was done when Zaifeng, Ruofan, and Dalin interned at Alibaba Group, advised by Zhen.

ACM Reference Format:
Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. 2023. RECom: A Compiler Approach to Accelerate Recommendation Model Inference with Massive Embedding Columns . In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3623278.3624761

1 INTRODUCTION

Deep recommendation model inference accounts for a large portion of data center workloads in many companies, including Google [8, 12], Meta [18, 40], and Alibaba [79, 80]. Typical deep recommendation models include two parts, which are the embedding layer and the deep neural network (DNN) stacks. In industry, the embedding layer consists of massive embedding columns (i.e., the subgraphs that transform input features to embedding vectors through table lookups) corresponding to different feature fields. It is common for developers to generate thousands of statistic features and process them with separate embedding columns for higher model accuracy [70]. However, processing such a large number of embedding columns is expensive. For example, our experiments on models in Alibaba show that embedding columns can be responsible for even more than 99% of the end-to-end inference latency on

GPUs. Therefore, optimizing the massive embedding columns in recommendation models is urgently needed. Meanwhile, ML compilers [34] are widely used to optimize conventional ML models automatically. But currently, none of them can optimize the heavy embedding columns. In this paper, we focus on using compilers to accelerate the recommendation model inference with massive embedding columns on the GPU.

Developing an ML compiler to accelerate the inference of deep recommendation models on the GPU is very important, as it can help companies save lots of manual effort. Traditionally, many companies have employed expert teams to optimize their recommendation models. However, there can be thousands of deep recommendation models in different business lines of a company with varied embedding column structures. Besides, due to privacy considerations, many real businesses require models to be optimized based on the intermediate representation (IR) of the computation graph rather than the source codes. For example, an industrial model in Alibaba contains 3.5 million lines of IR in TensorFlow GraphDef [1] format. Manually rewriting the model by analyzing such a large amount of IR is impractical, so an ML compiler approach is required to optimize the models automatically.

The characteristics of embedding columns in recommendation models bring three fundamental challenges for compilation optimizations. First, it is challenging to generate an efficient schedule on GPU for the massive operators within thousands of embedding columns. Traditional solutions cannot eliminate most of the large non-computation overhead and cannot fully utilize inter-subgraph parallelism. For a specific model (detailed in Section 7.6), XLA [15] can generate over 10,000 kernels, which introduces significant non-computation overhead and then results in only 33% of GPU active time. Moreover, most of the generated kernels have very small *waves per SM*[1] that less than 0.06. Second, the dynamic shape characteristics of recommendation models introduce many shape computation operations. The mixture of shape and tensor computations makes the graph topology too complicated to optimize. Third, the frameworks can introduce many redundant operations for robustness considerations. We found that the redundant computations can contribute to 80% of the entire embedding processing time on the GPU under extreme cases.

Researchers have put much effort into accelerating ML model inference [5, 15, 25, 26, 43, 78]. There are ML compilers [5, 15, 78] supporting automatic code generation with kernel fusion for lightweight memory-intensive operators, thus reducing the non-computation overhead caused by fragmented operators. However, these works target the conventional DNNs [31, 56], which cannot be applied to handle the new challenges emerging in the recommendation scenarios with massive embedding columns. Although several ad-hoc computation libraries [25, 26, 43] have been proposed to provide efficient implementations of specific operations like the embedding table lookup, they focus on optimizing only the matched patterns and require human efforts to rebuild the models using their interfaces. Therefore, no previous work provides full-scale automatic acceleration for recommendation model inference with massive embedding columns.

[1] *Waves per SM* [42] are calculated by dividing the launched block number by the total block number that can run concurrently on the GPU for the kernel.

In this paper, we propose **RECom**, an ML compiler that aims at accelerating the inference of deep recommendation models on the GPU. It effectively solves the challenges mentioned above. First, we develop a novel inter-subgraph parallelism-oriented fusion method to generate efficient code of the massive embedding columns in a single kernel. It identifies the subgraph corresponding to each embedding column, maps each subgraph to a group of threads on the GPU, and then fuses the computations within the subgraph together by leveraging the GPU hierarchical memory. Second, we develop a shape computation simplification module to address the shape computation challenge. It reconstructs all complex shape computations based on symbolic expressions. Third, to remove the redundant computations, we develop an embedding column subgraph optimization module to perform simplification by analyzing the graph contexts.

We evaluate RECom on four real-world in-house production recommendation models in Alibaba and two synthesized models. Experiments show that RECom outperforms state-of-the-art baselines by 6.61× and 1.91× in terms of end-to-end inference latency and throughput, respectively. In summary, this work makes the following contributions:

► We unveil the three significant performance challenges of embedding computations during recommendation model inference and offer a set of solutions and insights.

► We propose RECom, the first ML compiler that optimizes the time-consuming embedding column computations during the inference of recommendation models.

► We evaluate RECom on four industrial and two synthetic recommendation models. Experimental results show that RECom outperforms the baselines significantly.

2 BACKGROUND AND MOTIVATION

2.1 Characteristics of Recommendation Models

Deep recommendation models are widely used in various applications [8, 12, 40, 79, 80]. For example, e-commerce websites use them to recommend items to users by predicting user click-through rates (CTR) on potential items. Optimizing the inference of these models can bring significant economic benefits to businesses, as it enables them to predict more potential items in a limited time, thereby enhancing the quality of recommendations.

Deep recommendation model architecture. A deep recommendation model typically consists of two main components: the embedding layer and the DNN stack. The embedding layer often consists of multiple embedding tables and maps the input features into the low-dimensional embedding space [24]. The input features include user features (e.g., age and click history) and item features (e.g., price and category), which can be numerical values or strings. An embedding table is a lookup table that contains multiple learnable rows, and each row is called an embedding vector. The DNN stack usually contains attention structures [4, 79, 80] or multi-layer perceptrons (MLP). Figure 1 shows the architecture of a typical deep recommendation model for CTR prediction in the left part. During the inference of a single sample, the model takes in N input features and looks up the corresponding embedding vectors from the embedding tables. These embedding vectors are concatenated and fed into the DNN to predict the CTR.

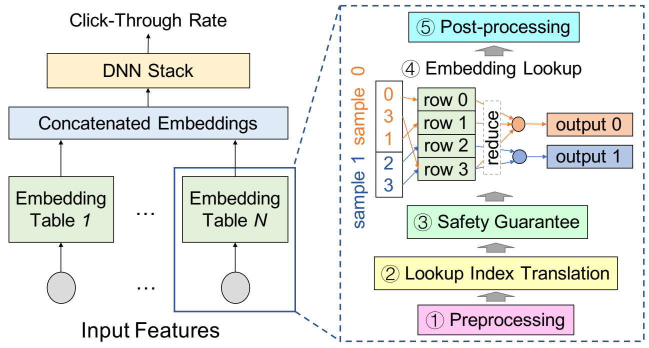


Figure 1: A typical deep recommendation model architecture. The dotted rectangle shows an embedding column example.

Embedding column. We define the subgraph that transforms an input feature into an embedding vector as an *embedding column*. The subgraph circled by the dotted rectangle in Figure 1 shows an example of the embedding column, which typically consists of five stages. ① *Preprocessing*. The embedding column first preprocesses the input features, mainly including string processing (e.g., string split) and numerical processing (e.g., multiplication). ② *Lookup index translation*. In this stage, the features are translated into indices for later embedding lookup using bucketing, hashing, or keeping the input data. ③ *Safety guarantee*. Several safety guarantee operations are inserted before embedding lookup to prevent potential errors, mainly including removing or replacing lookup indices that are out of bounds and filling a default value for samples with missing features. ④ *Embedding lookup*. In this stage, the embedding column retrieves rows from the embedding table according to the lookup indices. Multiple rows corresponding to the same sample are combined with element-wise reduction, as shown in Figure 1. ⑤ *Post-processing*. Finally, several post-processes can be applied to the embedding vector, such as replacing and removing partial values and reshaping.

During inference, the N input features are fed into the corresponding N embedding columns and go through the five stages to be transformed into the final embedding vectors. In production, the value of N, i.e., the number of embedding columns, can reach thousands, as there are many statistical features. More details of embedding columns are presented in Appendix A, including an example of building them and an explanation of why the number of them is typically large.

Dynamic shape. Existing compilers [5, 15, 78] rely on tensor shape information to optimize the model inference processes. However, due to *dynamic shapes*, we cannot determine the tensor shapes of recommendation models at compile time. *Dynamic shape* means that the shapes of tensors are dependent on the model inputs and can vary among different inferences. This characteristic is caused by operators whose output shapes depend on input contents rather than input shapes (e.g., *Where*[2]) and the dynamic input shapes (e.g., varied pooling factors [50] and batch sizes and absent features [50]).

[2] *Where* operator returns the indices of non-zero elements for a tensor. See https://www.tensorflow.org/api_docs/python/tf/where.

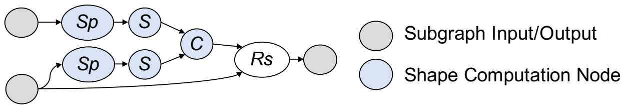


Figure 2: A subgraph from production models. Sp: *Shape*. S: *Slice*. C: *Concat*. Rs: *Reshape*.

2.2 Major Limitations of Existing Solutions

Several hand-written libraries [25, 26, 43] have been proposed to accelerate specific operations within embedding columns. However, they are difficult to deploy widely and have a limited application scope since they cannot enumerate all possible operator combinations to pre-build the library. Additionally, with numerous models and millions of lines of IR in each, it is impractical to optimize manually using these libraries.

In contrast, ML compilers can perform automatic optimizations on different models. They accept the computation graph IR, such as TensorFlow GraphDef [1] and ONNX [13], and then adaptively generate efficient codes for the target devices. The code generation process is also called *codegen*. Operator fusion is one of the essential technologies to reduce memory transactions and non-computation overheads during codegen [34]. Existing compilers like TVM [5] and XLA [15] adopt *input or output inline* (i.e., expanding the computation) to fuse *injective* (i.e., one-to-one map) operators to their consumers or producers. Unfortunately, existing ML compilers [5, 15, 37, 78] mainly focus on DNNs, ignoring the following performance challenges emerging in the massive embedding columns of recommendation models.

Challenge 1: Thousands of embedding columns with numerous small-sized operators make generating efficient schedules on the GPU difficult. For different subgraphs, the conventional fusion strategy of ML compilers either maps them into multiple kernels [5, 15] or executes each subgraph inside a kernel with global barriers [78]. These strategies, on the one hand, ignore the great parallelism among the subgraphs and cannot well utilize the GPU resources. Experiments with XLA [15] and AStitch [78] show that the *waves per SM of* most of their generated kernels are even less than 0.06, indicating a very low GPU utilization. On the other hand, they can lead to significant non-computation overhead due to frequent kernel launches and global synchronizations. Besides, the dynamic shape operators emerging in embedding columns (e.g., *Where* operator) pose new challenges in buffer management, further impeding the greater fusion granularity and exacerbating the non-computation overhead problem. Experiments for a model with 1,000 embedding columns (detailed in Section 7.6) show that applying XLA [15] can generate more than 10,000 separate kernels, resulting in only 33% GPU active time.

Challenge 2: Complicated shape computations in dynamic shape scenarios. *Shape computation* refers to the computation that operates on tensor shapes rather than tensor values. For example, Figure 2 shows a subgraph of shape computations with blue operators. The subgraph retrieves the shapes from two input tensors and then generates an output shape to feed the *Reshape* operator, which requires a shape input to reshape a tensor. In dynamic shape scenarios, more than 30% of the operators in embedding columns can be shape computations, as many computations require shapes as inputs for execution. Massive shape computations can break the tensor computation patterns, preventing many potential graph optimizations (detailed in Section 4.2).

Challenge 3: Inevitable redundant computations introduced by frameworks. Algorithm developers usually use high-level APIs provided by frameworks to build embedding columns (an example can be found in Appendix A.1). Due to robustness considerations, framework developers have to insert many safety guarantees within their functions. However, these operations can be redundant in the entire embedding column context. These redundant operations are inevitable from the framework aspect, as developers cannot be aware of the function usage context and cannot add any assumptions. Besides, algorithm developers tend to use general solutions for convenience rather than specific high-performance solutions, causing extra redundant overhead. Our experiments show that, under extreme cases, 80% of the computation time can be unnecessary for embedding columns.

RECom. In this paper, we propose a compiler system, RECom, to accelerate recommendation model inference by tackling the above three challenges with three components: *massive embedding column codegen* (Section 3), *shape computation simplification based on symbolic expressions* (Section 4), and *embedding column subgraph optimization* (Section 5).

3 MASSIVE EMBEDDING COLUMN CODEGEN

To tackle the first challenge in Section 2.2, we introduce the massive embedding column fusion and code generation approach (Section 3.1). Then, we describe the CPU-GPU co-running (Section 3.2) to expand hardware usage and reduce GPU memory usage.

3.1 Inter-Subgraph Parallelism-Oriented Fusion

We propose the *inter-subgraph parallelism-oriented fusion* approach to accommodate operators within thousands of embedding columns into a single GPU kernel. This approach can significantly reduce the non-computation overhead of embedding columns and exploit both intra- and inter-subgraph parallelism.

3.1.1 Inter-Column Parallelism Mapping. To generate codes with high parallelism, RECom maps the independent embedding columns to different groups of GPU threads. The threads within each thread group process one embedding column cooperatively and leverage the hierarchical memory for intermediate data buffering and inter-thread communication. In this way, RECom can effectively exploit both intra-operator and inter-column parallelism of recommendation models with the GPU.

Specifically, we use a thread block to process one embedding column. As the executions of different thread blocks are independent, the pattern divergence between embedding columns performs well. Besides, the tail effect between embedding columns can be compensated by the thread block scheduling. As discussed in Section 2.1, the number of embedding columns can be more than one thousand in modern recommendation models, while the workload of each column is not heavy (batch size is often hundreds). At the same time, the most popular GPUs used for ML inferences, e.g., NVIDIA T4 and A10, can accommodate hundreds to around one thousand thread blocks concurrently. Therefore, mapping each embedding column to a thread block leads to a good map between the model parallelism and GPU resources.

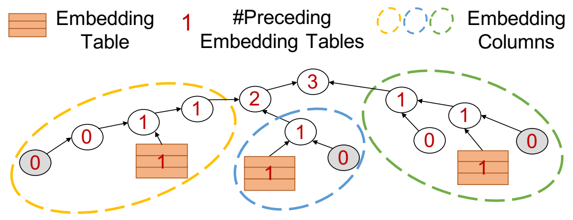


Figure 3: Illustration of embedding column identification.

Given the basic insight of inter-column parallelism mapping, the obstacles to enabling compilation optimization mainly includes embedding column identification (Section 3.1.2), code generation of every single embedding column containing complex patterns (Section 3.1.3), and buffer allocation without knowing tensor shapes at compile time (Section 3.1.4).

3.1.2 Domain Knowledge-based Embedding Column Identification. In a recommendation model with massive embedding columns, it is often unclear which operator belongs to which specific column. This is because real businesses usually require optimizing the models with only computation graph IR, which provides little information for partitioning different embedding columns. This lack of clarity poses a significant obstacle to *inter-embedding-column parallelism mapping*, which requires a clear partitioning of the columns.

Fortunately, we make several important observations that enable effective partitioning of embedding columns in a computation graph. First, in typical recommendation models, the trainable variables in the graph are typically network weights, biases, or embedding tables. Gathering operations are often applied to the embedding tables according to the function of embedding columns. Meanwhile, compute-intensive network operators like GEMM and convolution are usually applied to the network weights, and element-wise addition is applied to the biases. Based on this observation, RECom identifies the variables that maintain the embedding tables rather than the network weights or biases. Second, in deep recommendation models, one embedding column typically corresponds to one embedding table.

With the above domain knowledge of embedding columns, we can cluster the operators that consume only one embedding table variable directly or indirectly, as well as all their preceding operators, into a subgraph corresponding to that embedding table. Operators that consume multiple embedding tables do not belong to an embedding column subgraph. These operators include the converging points of multiple embedding columns (e.g., *Concatenate*) and their following neural network operators. The above approach is illustrated in Figure 3. For each operator, we first count the number of embedding tables it consumes, presented as the red numbers in the figure. Then starting from the embedding table variables, we cluster all the different embedding columns, which are circled by different colors in the figure. Appendix B illustrates the detailed procedure of embedding column identification.

3.1.3 Group-based Intra-Column Code Generation. Unlike ML compilers focusing on regular tensor computations such as GEMM and

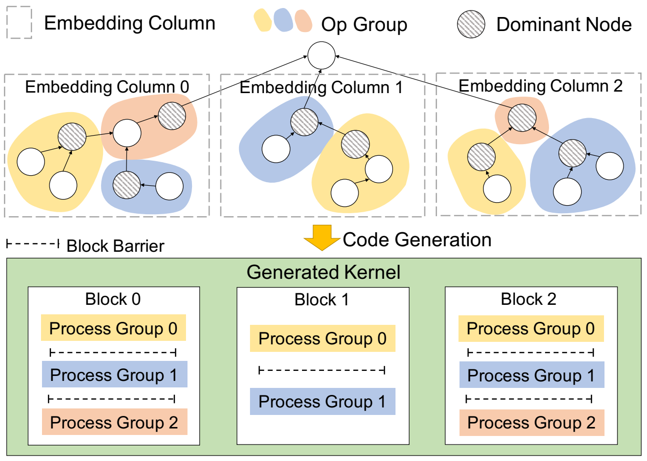


Figure 4: Illustration of the code generation procedure with a sample 3-embedding-column graph.

element-wise operations, RECom targets the embedding column computations involving many distinct operations. These operations include shape computations, dynamic-shape operations such as the *Where* operator, and embedding lookup operations [39].

To accommodate all the complex computations within a single kernel, RECom uses a divide-and-stitch approach for code generation. It divides the operators of the embedding column subgraph into several groups according to specific rules (described in the next paragraph), generates the code of each group independently, and "stitches" the groups together inside the same GPU kernel by buffering the intermediate data on either shared memory or global memory. The code of each embedding column is generated independently and mapped to specific thread blocks.

Different from existing compilers [5, 15, 78], RECom generates code for both tensor and shape computations in the fused kernel. For tensor computations, it borrows the basic insight of grouping and stitching approach in AStitch [78]. RECom identifies all the non-*injective* operators [5], which we call *dominant* operators. RECom clusters each *dominant* operator with its *injective* producers to form an operator group for code generation. It generates the code of the *dominant* operator by the corresponding template and expands the computations of *injective* operators inline within a group. To stitch the groups together, intermediate data between groups is buffered into shared memory if its shape can be determined small enough at compile time; otherwise, it is buffered into global memory. RECom inserts block-level thread barriers to ensure data coherence between groups. For shape computations, RECom simplifies each complex shape computation subgraph by reconstructing it with a unified operator of symbolic expressions, which will be discussed in Section 4.2. RECom then translates the symbolic expressions directly into code for shape computation. RECom performs shape computations independently in the GPU registers for all threads in the block rather than using shared memory to broadcast results. This is because the shape computations are usually very lightweight compared to extra memory transactions and block synchronizations. The results of the shape computations are then made inline into the consumer operators.

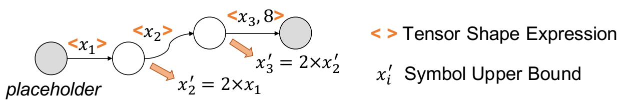


Figure 5: Illustration of the buffer allocation by inferring the memory upper bounds. x'_i denotes the upper bound of x_i.

Figure 4 illustrates the code generation procedure of a graph with three embedding columns. For each embedding column, RECom first identifies the dominant operators and forms an operator group as the code generation unit. Then, for the groups within an embedding column, RECom "stitches" the generated codes for both tensor and shape computations with barriers inserted and locates them in a block.

3.1.4 Compile Time and Runtime Combined Buffer Allocation. RECom needs to pre-allocate the global memory buffer of the outputs and the intermediate data for the generated kernel before launching it. However, output shapes for many operators of embedding columns can only be determined after execution. For example, the output tensor shape of the *Where* operator is determined by the content of the tensor value rather than the input shapes. A naïve approach is to allocate buffers inside the fused kernel at runtime dynamically. However, runtime allocation inside the kernel is quite expensive.

We design the *compile time and runtime combined buffer allocation* approach to solve this problem. The key idea is to infer the tensor shape upper bound expression of each operator at compile time and allocate the global memory buffer accordingly at runtime. Specifically, RECom infers the upper bound of the shape for each tensor at compile time with the help of symbolic shape expressions. Since the embedding column operators typically have dynamic shapes, RECom cannot determine the exact shape value at compile time. Instead, RECom represents the shape of each tensor with symbolic expressions, which will be discussed in Section 4.1. Every time a new symbol is added during the shape inference, RECom calculates the upper bound of the symbol based on the existing symbols. As shown in Figure 5, the left input operator is a placeholder with shape $<x_1>$, and subsequent operators generate new symbols x_2 and x_3. RECom calculates the upper bound of these symbols based on the operators used. Using the symbol upper bound and the tensor shape expressions, RECom generates the corresponding host code to compute the buffer size required for allocation at runtime based on the input shape $<x_1>$. To reduce the overheads due to frequent memory allocations for the massive embedding columns, RECom merges all required buffers into a single large buffer that can be allocated at once. Experiments show that merging the buffer allocations reduces execution time by 35.1% (detailed in Section 7.5).

Although the graph has dynamic shapes, the contents of partial buffers can be determined statically at compile time. For example, the *Bucketize*[3] operator requires a constant buffer to store the boundary values. To eliminate allocation and assignment overheads for such buffers, RECom allocates and assigns them during graph construction rather than execution.

[3] *Bucketize* bucketizes the input tensor based on the given boundaries. See https://www.tensorflow.org/api_docs/python/tf/raw_ops/Bucketize .

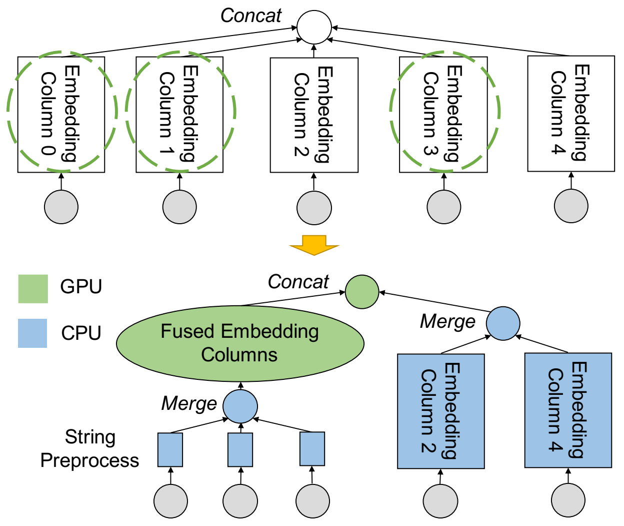


Figure 6: An example of CPU-GPU co-running.

3.2 CPU-GPU Co-Running

RECom excludes two kinds of operations of embedding columns from GPU fusion and instead executes them on the CPU. The first is the string operation (e.g., regular expression matching) in the string preprocess stage, which is typically better suited for CPU processing. The second is the embedding column with an embedding table size that exceeds a specific threshold. These large tables typically correspond to features in the ID category (e.g., user ID and item ID), which make up only a small percentage of the total features. For example, only five of the 1,277 columns of model A in Section 7 have embedding tables larger than 256MB. Placing these few columns on the GPU provides only a slight performance improvement but consumes much more GPU memory. By enabling CPU-GPU co-running, RECom can enable a single GPU to handle models with parameter sizes larger than its memory capacity. Additionally, offloading these two types of operations to the CPU can help better utilize the CPU computing resources.

After processing the embedding columns on both the CPU and GPU, RECom concatenates all their outputs in the order of the embedding columns to feed the subsequent DNNs. As GPUs are ideal for processing compute-intensive operators in the DNN part, all the embedding column outputs on the CPU need to be transferred to the GPU. However, moving the CPU data of different embedding columns one by one can cause significant performance degradation due to the overhead of PCIe bus transfers and data transfer latency. To address this issue, RECom merges these outputs on the CPU into a contiguous array and then transfers it to the GPU memory in one shot. Similarly, RECom merges the string preprocess results on the CPU and then moves them to the GPU at once to feed the fused kernel. The merging of cross-device transfers results in a 4.4× speedup, as detailed in Section 7.5.

Figure 6 demonstrates the co-running of a model with several embedding columns. RECom fuses the circled embedding columns into one GPU kernel while leaving the string processes and columns with large tables on the CPU. Tensors on the CPU are merged before being transferred to the GPU.

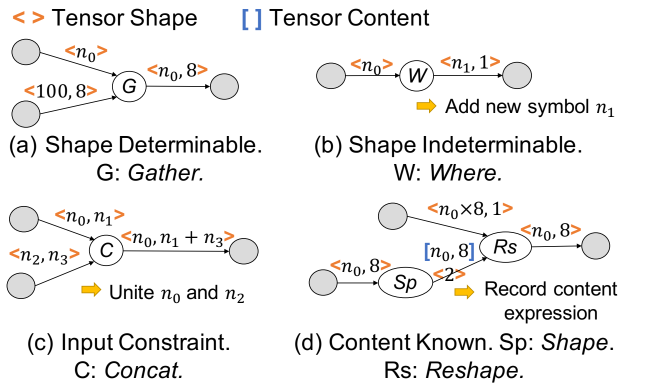


Figure 7: Four pattern abstractions of symbolic shape expression inferences.

4 SHAPE COMPUTATION SIMPLIFICATION BASED ON SYMBOLIC EXPRESSIONS

To address Challenge 2 of shape computations discussed in Section 2.2, we propose an approach based on symbolic expressions in this section. Similar to BladeDISC [77], RECom builds the global symbolic shape expressions of the embedding columns (Section 4.1). Based on the symbolic expressions, we propose simplification methods through shape computation reconstruction (Section 4.2). Additionally, we show the elimination of shape-only tensor computations, which is a minor optimization, in Appendix C.

4.1 Global Symbolic Shape Expression Inference

The lack of shape information under dynamic shape scenarios makes further graph optimizations difficult. To simplify shape computations and enable other optimizations (Section 3.1 and Section 5.2), RECom uses symbol-based expressions to represent the shape of each tensor, which is called "symbolic shape expression". For example, a tensor's shape can be represented by the symbolic expression $<n_1 + n_2, 8>$, where the values of symbols n_1 and n_2 are determined at runtime. Given a computation graph of the recommendation model, RECom infers the symbolic shape expression from the input to all operators.

Figure 7 illustrates the four symbolic shape propagation patterns: *shape determinable*, *shape indeterminable*, *input constraint*, and *tensor content known*, respectively. In Figure 7 (a), the *shape determinable* pattern is shown, where the output shape expressions can be directly inferred from the input shapes. In Figure 7 (b), the *shape indeterminable* pattern is depicted, where the output shape expressions cannot be inferred without the content of the input tensors. In such cases, RECom assigns a new symbol to the symbol table and uses it as an unknown dimension size for the output shape. Figure 7 (c) demonstrates the input constraint pattern. In the *Concat* operator, the sizes of the dimensions to be concatenated should be equal. Hence, RECom unites the symbols n_0 and n_2 in Figure 7 (c). Finally, Figure 7 (d) presents an example of *tensor content known*

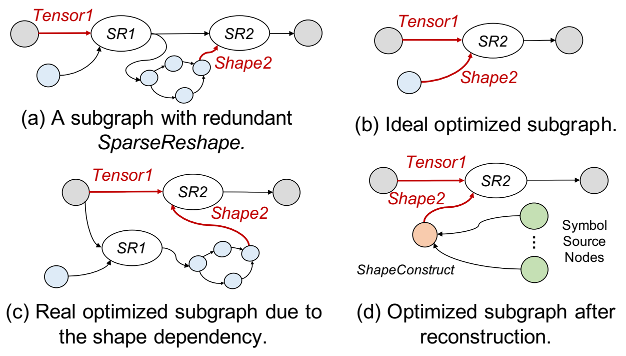

Tensor1
SR1
SR2
Shape2
(a) A subgraph with redundant *SparseReshape.*
Tensor1
SR2
Shape2
(b) Ideal optimized subgraph.
Tensor1
SR2
Shape2
SR1
(c) Real optimized subgraph due to the shape dependency.
Tensor1
SR2
Shape2
Symbol Source Nodes
ShapeConstruct
(d) Optimized subgraph after reconstruction.

Figure 8: Example of how the shape computation reconstruction eases the further graph optimizations. Operators in light blue represent the shape computations.

pattern, where an operator requires a tensor to express the shape of the output tensor, as in the case of the *Reshape* operator. In this situation, RECom extracts the shape tensor content to help build the shape expression for the output tensor.

4.2 Unified Shape Computation Reconstruction

The complex coupling of tensor and shape computations makes graph optimization difficult. To ease the graph optimizations, RECom reconstructs shape computations based on the symbolic expressions of the outputs. Specifically, RECom substitutes each subgraph of shape computation with a unified *ShapeConstruct* operator. The *ShapeConstruct* operator performs the shape computation logic according to the corresponding symbolic expression.

By reconstructing the shape computations, RECom simplifies shape computation subgraphs and isolates them from the tensor computations. Figure 8 demonstrates how the decoupling of tensor and shape computations simplifies the graph and facilitates further graph optimizations. Figure 8 (a) shows a subgraph of two connected *SparseReshape* operators. If the only output of *SR1* is *SR2*, then *SR1* is actually redundant because the two *SparseReshapes* can be merged into one logically, as shown in Figure 8 (b). However, simply reconnecting the *Tensor1* and *Shape2* onto the latter *SparseReshape* operator results in the case in Figure 8 (c), where *SR1* is still present. This is because the tensor *Shape2* still indirectly relies on the output of *SR1*. Nevertheless, this dependency is unnecessary as the value of *Shape2* does not depend on the computation of *SR1*. Fortunately, RECom can remove this dependency after shape computation reconstruction. This is because the replaced *ShapeConstruct* operators only depend on operators who really generate the required symbols, as shown in Figure 8 (d). Consequently, the useless *SR1* can be eliminated completely.

5 EMBEDDING COLUMN SUBGRAPH OPTIMIZATION

In this section, we focus on the graph-level optimizations on each embedding column subgraph to tackle the challenge of redundant computations discussed in Section 2.2. We present how RECom detects and eliminates unnecessary safety guarantees (Section 5.1) and simplifies embedding lookup procedures (Section 5.2).

5.1 Unnecessary Safety Guarantee Elimination

For robustness consideration, framework developers have to insert several safety guarantees before embedding lookup (as shown in Figure 1). However, upon analysis of the entire embedding column subgraph, we observe that most of these operations are actually redundant and can be eliminated without posing any safety risks or impacting the inference results.

Lookup index removal. Some safety guarantees are typically applied to remove out-of-range lookup indices before embedding lookup to prevent potential crashes. However, these guarantees are often unnecessary as the lookup index translation stage already maps the input data into valid ranges. Due to the lack of knowledge of the runtime context and graph construction, framework developers cannot determine whether such index removals are necessary, resulting in redundant ones. These operations can consume up to 17.7% of TensorFlow-GPU execution time (detailed in Section 8.5). We propose an index range analysis and propagation approach to address this issue at compile time. We identify frequent index operation patterns in numerical preprocess, index translation, and safety guarantees of embedding columns and use them to propagate index value ranges at each stage. If the index value range already satisfies lookup requirements after translation, RECom removes the redundant safety guarantees of index removal. More details of the above process are shown in Appendix D.

Empty row filling. Empty row filling is used to fill a default lookup index for samples whose feature is absent for a certain feature field, which is quite heavy on the GPU. This operation guarantees that each sample has at least one lookup index. However, it is common for the corresponding embedding vectors of these empty rows to be replaced with zeros during the post-processing stage. In such cases, RECom proposes a modified lookup operator that merges the empty row filling, embedding lookup, and post-replacement stages. This new operator directly sets the empty rows' corresponding vectors to zeros, thus eliminating the need for expensive empty row filling and bringing 1.49× speedup over TensorFlow-GPU (detailed in Section 7.5).

5.2 Embedding Lookup Simplification

The embedding lookup in the embedding column contains two steps, which are the embedding table lookup by input indices and the embedding vector reduction for the same sample [39].

However, the reduction operation can also be redundant if the corresponding feature is a one-hot feature (i.e., univalent [12]). As the framework developers cannot determine if the input is one-hot or multi-hot when designing the lookup function, they treat all features as multi-hot (i.e., multivalent [12]) to handle all input cases. This design can lead to sub-optimal solutions for one-hot features. RECom can detect this problem by using symbolic shape expressions and prune the redundant reduction operations.

Besides, some algorithm developers may assign default values to absent features, such as an empty string for string features. In these cases, if the corresponding feature is also one-hot, the lookup indices become dense tensors rather than sparse. RECom identifies

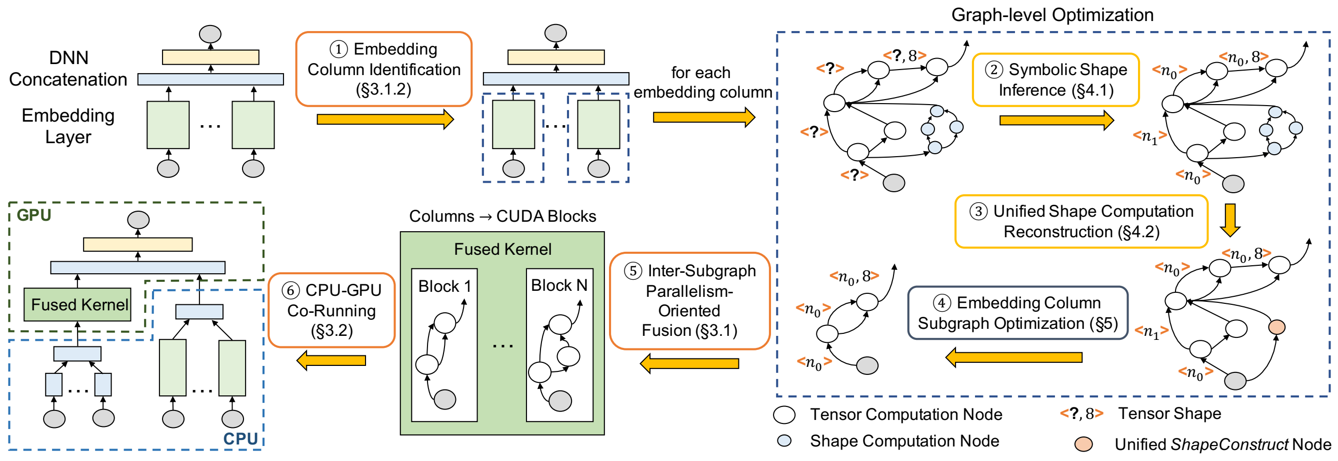

Graph-level Optimization
DNN Concatenation
Embedding Layer
① Embedding Column Identification (§3.1.2)
for each embedding column
② Symbolic Shape Inference (§4.1)
③ Unified Shape Computation Reconstruction (§4.2)
④ Embedding Column Subgraph Optimization (§5)
⑤ Inter-Subgraph Parallelism-Oriented Fusion (§3.1)
⑥ CPU-GPU Co-Running (§3.2)
Columns → CUDA Blocks
Fused Kernel
Block 1
Block N
GPU
CPU
Fused Kernel
Tensor Computation Node
Shape Computation Node
<?, 8> Tensor Shape
Unified ShapeConstruct Node

Figure 9: The optimization workflow of RECom. In the graph-level optimization (②-⑤), we only show the shapes of partial tensors for illustration.

Table 1: Statistics of models in the experiments.

Model	Source	# Output Targets	# Embedd. Tables	Embedd. Dim.	Embedd. Dim. Sum	Table Size	Embedd. Param. Size	DNN Param. Size
A	Production	7	1277	4-20	9516	16B-1.0GB	4.6GB	47MB
B	Production	7	1155	4-20	8596	32B-1.1GB	1.7GB	40MB
C	Production	9	816	4-20	5960	16B-2.5GB	3.8GB	29.4MB
D	Production	6	1050	4-20	7768	16B-1.35GB	2.6GB	35.1MB
E	Synthesized	1	1000	8-32	8120	3.1KB-1.0GB	5.1GB	36.4MB
F	Synthesized	1	1000	8-32	9672	3.1KB-1.0GB	3.1GB	89.8MB

this situation by verifying the symbolic shape expressions and then transforms the tensors from sparse formats into dense formats. Furthermore, RECom replaces the related sparse operators with more efficient ones targeting dense tensors.

By simplifying each lookup procedure accordingly, we observe a 1.86× speedup over TensorFlow-GPU (detailed in Section 7.5).

6 IMPLEMENTATION

Optimization workflow of RECom. Figure 9 shows the optimization workflow of RECom. Given a computation graph of a recommendation model, RECom performs the following optimization steps. ① RECom first identifies the embedding columns in the model based on our domain knowledge. ② For each embedding column, as the shape information is missed, RECom performs symbolic shape inference to represent tensor shapes as symbolic expressions. ③ Based on the symbolic expressions, RECom replaces each shape computation subgraph with a unified operator to simplify the computation graph. ④ With the simplified graph, RECom further eliminates the redundant computations based on the analysis of the entire embedding column subgraph. ⑤ After graph-level optimization for each column, RECom fuses the columns into a single GPU kernel and maps each column into a thread block. ⑥ Finally, RECom places CPU-friendly operations on the CPU to enable CPU-GPU co-running.

Implementation details. Currently, we implement RECom as a TensorFlow [1] add-on consisting of 12.5K lines of C++ code. RECom utilizes *SymEngine library* [54] to perform symbolic expression computations in Section 4. RECom leverages the TensorFlow API to register custom operators and graph optimization passes. To use RECom, users only need to add one line in the script to load the dynamic-link library, without modifying the source code of neural networks. Then, RECom can obtain the original computation graph and replace it with an optimized one. Although we implement it as a TensorFlow add-on, the basic idea can be applied to other ML frameworks as well.

7 EVALUATION

7.1 Experimental Setup

Models. In our experiments, we use four real-world in-house production recommendation models and two synthesized models to evaluate RECom. The model statistics are listed in Table 1. The four production models are from Alibaba, and their model types are Deep Bayesian Multi-Target Learning (DBMTL) [58]. We also synthesize two Deep Learning Recommendation Models (DLRM) [40] with different configurations and the corresponding feeding data for reproducibility. We use TensorFlow *FeatureColumn* API [1] to build the embedding columns for these two models.

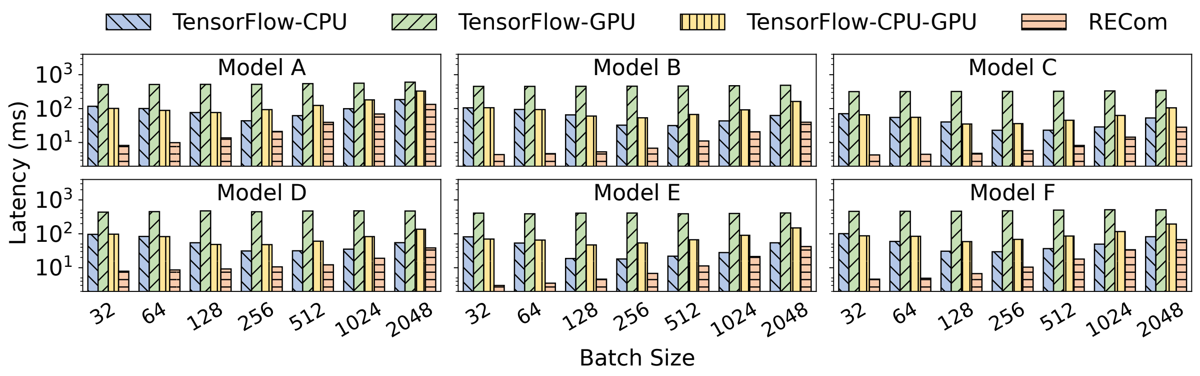


Figure 10: Given the same batch sizes, RECom achieves speedups of 6.61×, 51.45×, and 8.96× for average inference latency compared with TF-CPU, TF-GPU, and TF-CPU-GPU, respectively. The vertical axes are latency in the log scale.

Baselines. For the end-to-end comparison in Section 7.2, we use TensorFlow (TF) [1] with different device configurations as our baselines, including TF-CPU, TF-GPU, and TF-CPU-GPU. This is because no existing work can automatically optimize the heavy embedding columns in current recommendation models, as discussed in Section 2.2. For TF-CPU-GPU, we put the embedding layer on the CPU and the DNNs on the GPU to accelerate the DNN stacks. Besides, we perform further case studies to compare with ML compilers (XLA [15] and AStitch [78]), ad-hoc libraries (NVIDIA HugeCTR [43]), workload schedulers (DeepRecSys [17]), and our internal manually-optimized solutions in Section 7.6.

Due to fairness considerations, we use different numbers of CPU cores between experiments on the CPU-GPU platform and on the CPU-only platform. We use **4** CPU cores when GPU is used (RECom, TF-GPU, and TF-CPU-GPU) and **32** CPU cores for pure CPU experiments (TF-CPU) according to their prices on Alibaba Cloud platform.

Hardware/software specifications. We perform experiments on machines equipped with an Intel Xeon Platinum 8163 CPU and an NVIDIA Tesla T4 GPU, whose TDPs are 150W and 70W, respectively. NVIDIA Tesla T4 is a popular inference card due to its abundant low-precision computing resources, low energy usage, and competitive price. The TensorFlow version is 2.6.2, with OneDNN optimization [23] enabled. Our codes are compiled by GCC 7.3 and NVCC 11.3 with -O3 enabled.

Metrics. We evaluate the latency and the throughput under service level agreements (SLA) of RECom to demonstrate its effectiveness, as they are both important metrics for model inferences in production environments [18].

Configuration. To measure the inference latency, we launch a TensorFlow session to process 1,000 queries sequentially after warming up and then calculate the average latency. We prepare all the input features in advance and convert them into TensorFlow tensors before the measurement. As for throughput measurement, we launch a session with multiple worker threads to serve 1,000 queries concurrently. The batch sizes of the queries are tuned to meet the SLA constraint.

7.2 End-to-End Performance

Latency under different batch sizes. Figure 10 presents the inference latency of RECom, TF-CPU, TF-GPU, and TF-CPU-GPU on the six models, with batch sizes ranging from 32 to 2048. Experiments show that for all models under any batch size, RECom outperforms the three TensorFlow baselines significantly. On average, RECom achieves speedups of 6.61×, 51.45×, and 8.96× for inference latency compared with TF-CPU, TF-GPU, and TF-CPU-GPU, respectively.

An interesting observation is that, under small batch sizes, the inference latency of TF-CPU does not strictly increase as batch size increases. The reason is that TensorFlow tends to schedule cheap operators on a single thread so that a large number of operations in different embedding columns can be executed sequentially under small batch sizes.

We observe that TF-GPU brings much higher latency than TF-CPU for all batch sizes. This is because the thousands of embedding columns introduce massive small operators, leading to significant non-computation overhead and hardware under-utilization. We show the detailed analysis in Section 7.4. The latency gap between TF-GPU and TF-CPU decreases as the batch size increases. This phenomenon is in line with our expectations. On the one hand, larger batch sizes increase the GPU speedup of DNN over the CPU. On the other hand, due to the GPU under-utilization problem of TF-GPU, the execution latency of the embedding layer is insensitive to the batch size.

TF-CPU-GPU does not improve performance compared with TF-CPU. On the one hand, the computations of DNNs are lightweight compared with embedding columns, so putting them on GPU brings little benefit. On the other hand, for a fair comparison, the number of CPU cores used for TF-CPU-GPU is less than that for TF-CPU. This causes higher latency on the embedding computations. Besides, the memory copy from the CPU to the GPU introduces extra overhead.

Throughput under SLA. In Figure 11, we present the throughput (inferences per second) comparison between RECom and TF-CPU under the SLA of 100ms with different numbers of worker threads. The results show that RECom achieves 1.91× improvement over TF-CPU. We do not present the throughput of TF-CPU-GPU and TF-GPU because they either cannot satisfy the SLA, whatever the batch size is, or can only achieve little throughput.

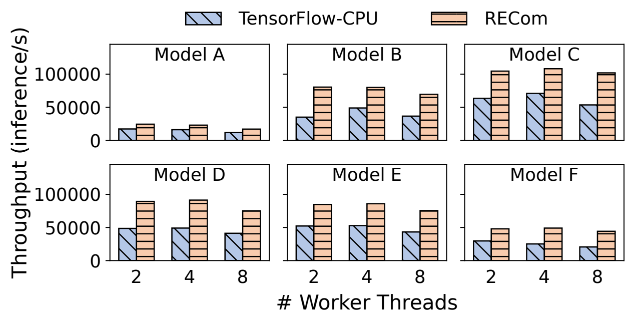
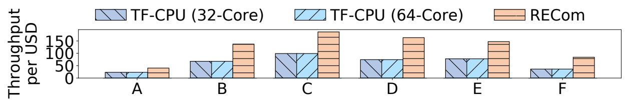
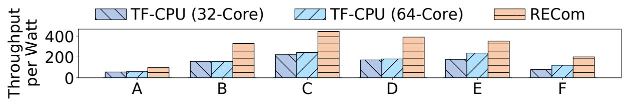


Figure 11: Under SLA of 100ms, RECom improves the throughput by 1.91× over TF-CPU. TF-CPU-GPU and TF-GPU are not presented as they cannot meet the SLA.



(a) Throughput normalized to one-month cloud rental fee.



(b) Throughput normalized to energy cost.

Figure 12: For throughput normalized to price and energy, RECom achieves 2.01× and 1.97× average improvement over TF-CPU, respectively.

7.3 Normalized Throughput

As RECom and the TF-CPU baseline use different hardware resources (T4 + 4 CPU cores for RECom, and 32 CPU cores for TF-CPU), we present the throughput normalized to the resources with four worker threads in this section to prove the fairness of our comparisons. We also add the evaluation with 64 CPU cores for TF-CPU with eight workers in this section.

Cloud instance prices. Our experiments are performed on three popular cloud instances provided by Alibaba Cloud, which are `ecs.c5.8xlarge`, `ecs.gn6i-c4g1.xlarge`, and `ecs.g5.8xlarge`. The rental price of a cloud instance is a crucial factor in reflecting the resource cost of each solution, as cloud vendors determine the prices of different instances by considering factors like hardware and energy costs. On the other hand, for companies deploying their models on the cloud, renting a cloud instance is the only cost they need to consider.

Figure 12 (a) shows the comparison of throughput normalized to the rental cost of RECom and TF-CPU. The results are figured out by dividing the throughput by the corresponding one-month cloud rental fee. We find that at the same rental cost, RECom obtains an average of 2.01× throughput compared with TF-CPU.

Energy costs. We also compare the energy cost of RECom and TF-CPU. By directly dividing by the TDP of the platform, RECom still achieves 1.3× normalized throughput compared with TF-CPU. However, as RECom only uses the CPU's partial resources (4 cores), TDP cannot accurately reflect its energy consumption. Therefore, we use `Intel PCM` [22] and `nvidia-smi` [44] to monitor the CPU's and GPU's actual power usage during the evaluation. We sample the power consumption every 0.5 seconds and take the average power after finishing the execution.

Figure 12 (b) presents the throughput normalized to the energy cost of RECom and TF-CPU. Although using more cores for TF-CPU increases its throughput, the normalized throughput does not change significantly. Experimental results show that at the same energy cost, RECom achieves a 1.97× speedup over TF-CPU in terms of throughput.

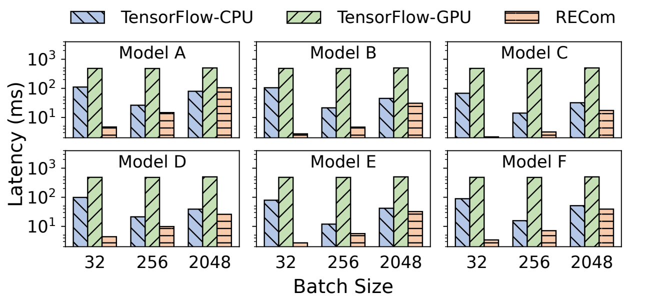


Figure 13: On average, RECom achieves 9.89× and 74.17× speedup of the embedding latency over TF-CPU and TF-GPU.

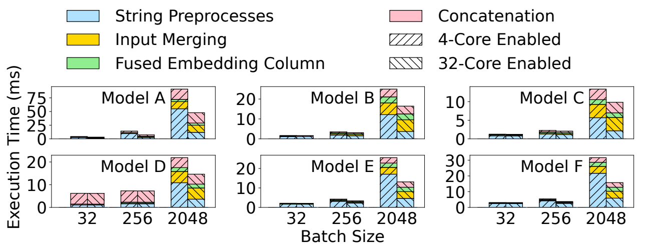
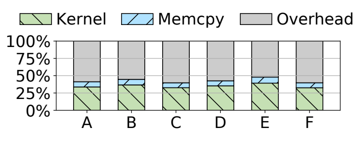


Figure 14: Embedding layer breakdown of RECom equipped with different numbers of CPU cores. The left bar with "//" indicates 4 CPU cores are enabled, while the right bar with "\\" indicates 32 cores are enabled.

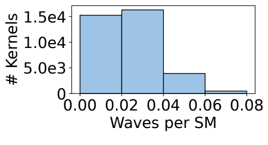
7.4 Breakdown

Embedding-only inference latency. To evaluate the performance improvement on the embedding column processing of RECom, we remove the DNNs in the models and measure the latency. Figure 13 presents the embedding-only inference latency of RECom, TF-CPU, and TF-GPU with different batch sizes. The results show that, on average, RECom achieves 9.89× and 74.17× speedups of the embedding latency over TF-CPU and TF-GPU.

Embedding column performance breakdown. We observe that as the batch size increases, the speedup of RECom decreases. The decrease is because the string preprocessing on the CPU with a few cores cannot keep up with the processing speed of the GPU



(a) Execution time breakdown.



(b) Waves per SM histogram.

Figure 15: Performance analysis of TF-GPU.

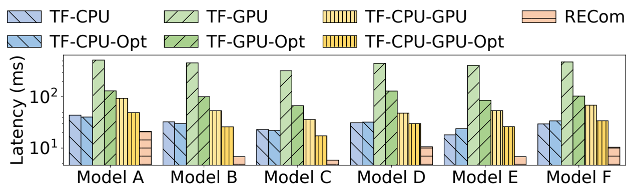


Figure 16: End-to-end latency comparison before and after graph-level optimizations being applied with the batch size of 256. The "-Opt" suffix indicates that the graph is optimized by shape computation simplification and embedding column optimization.

consumer at large batch sizes. We present the execution time of string preprocess (CPU), input merging and transmission (CPU-GPU), fused embedding column processing (GPU), and embedding vector concatenation (GPU) with different batch sizes in Figure 14. When batch size is 2,048, the string preprocessing can account for up to 60.5% of the total processing time with the 4-CPU-core setting. However, by increasing the number of CPU cores to 32, we can observe up to 78.3% reduction of the string preprocessing overhead. Note that before the optimization of RECom, even for model A under the batch size of 2,048, the string preprocessing time accounts for only 19% of the total execution time on the CPU.

TF-GPU performance breakdown. Section 7.2 shows that unlike traditional models (e.g., CNNs and Transformers), the performance of TF-GPU is significantly worse than TF-CPU for recommendation models. We present the execution time breakdown of TF-GPU in Figure 15 (a) to reveal the reasons behind its low performance. We use the batch size of 256 in this evaluation and only show the breakdown of the embedding parts. We find that the GPU kernel execution time accounts for only 35% of the entire embedding execution time on average. This is because kernel launches, synchronization, and operator scheduling introduce tremendous overheads. For example, during the execution of the embedding parts of model A, there are 136,644 operators and 37,580 GPU kernels. Although the overhead of each kernel launch, kernel synchronization, and operator scheduling is around several microseconds (e.g., launch overhead is 1.08μs as reported in [69]), the accumulated overhead is huge and accounts for 59% of the execution time. By fusing the numerous operators across embedding columns, RECom eliminates most of the above overhead.

Besides, most of the kernels in TF-GPU are too small-scaled to utilize the resources on the GPU fully. Figure 15 (b) shows the distribution of *waves per SM* of TF-GPU kernels on model A. *Waves per SM* [42] (i.e., wave number) reflect the GPU resource utilization by taking all of the launched blocks, SM number, and occupancy into consideration. We omit the kernels with wave numbers higher than 0.08 as they account for only less than 5% of total kernels. As shown in Figure 15 (b), the wave numbers of 94.4% of kernels are even less than 0.06, indicating significant waste of GPU resources. By contrast, the fused kernel of RECom achieves an average wave number of 2.43 in our evaluation.

7.5 Ablation Study

Impact of graph-level optimizations and massive embedding column codegen. Both modules of shape computation simplification and embedding column optimization (including unnecessary safety guarantee elimination and lookup procedure simplification) perform graph-level optimizations. Experimental results show that the graph-level optimizations reduce 84.42% of the operators in the embedding columns. We then present the inference latency of raw TF, only graph-level optimizations enabled, and full RECom optimizations enabled in Figure 16 under the batch size of 256. As these graph-level optimizations are independent of specific hardware, we apply the graph-level optimizations to all TF-CPU, TF-GPU, and TF-CPU-GPU. We use the "-Opt" suffix to indicate that graph-level optimizations are performed. For example, TF-CPU-GPU-Opt means the graph-level optimizations are applied, with the embedding parts on the CPU and DNNs on the GPU. Compared with them, RECom further enables the massive embedding column fusion and codegen optimization.

The results show that by enabling graph-level optimization, we can bring 4.37× speedup on TF-GPU. In detail, the shape computation simplification and embedding column optimization contribute 1.22× and 3.58× speedups, respectively. For the embedding column optimization, eliminating the unnecessary invalid index removals and empty row fillings brings 1.22× and 1.49× speedups, respectively. Simplifying each embedding lookup procedure also brings a 1.86× speedup. By applying the column fusion and code generation after graph-level optimization, we can further achieve an 11.20× speedup. The massive embedding column codegen contributes to the major performance improvement of RECom. By disabling the buffer allocation merging and the CPU-GPU data transfer merging during codegen, we observe 1.54× and 4.47× of performance degradation, respectively.

The graph-level optimizations also bring a 1.95× speedup on TF-CPU-GPU, which is smaller than that on TF-GPU. On the one hand, the large number of operators can introduce more significant overhead on the GPU due to frequent kernel launches, synchronizations, and off-chip memory accesses. On the other hand, redundant computations like *Unique* operators are more expensive on the GPU than on the CPU.

TF-CPU gains little from the graph-level optimizations because the 32 cores are not fully utilized, and the scheduling policy of TensorFlow can make TF-CPU-Opt even slower than TF-CPU.

Memory saving from CPU-GPU co-running. We put embedding columns with large embedding tables (≥ 256MB) on the CPU, as the number of such columns is often small but consumes a large memory footprint. As shown in Table 2, several embedding tables can account for up to 97.4% of the entire model parameters in

Table 2: The information of large embedding tables.

Models	A	B	C	D	E	F
#Large Tables (≥ 256MB)	5	2	4	3	5	3
Memory Consumption (GB)	4.2	1.36	3.7	2.41	5.0	3.0
Percentage of All Tables (%)	91.3	80.0	97.4	92.7	98.0	96.8

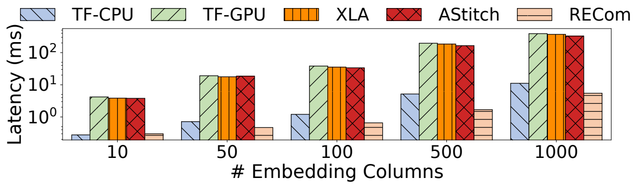


Figure 17: Comparison between RECom and TF baselines and other ML compilers with different numbers of columns.

production. Experimental results indicate that by offloading these columns to the CPU, the inference performance only changes within 5% on average, but 92.7% of the GPU memory usage can be saved.

7.6 Microbenchmark Comparison with Prior Works

ML compilers like XLA and AStitch. We synthesize models with various numbers of typical embedding columns (from 10 to 1,000) and measure the latency of their embedding parts. We believe this range of embedding column numbers can cover most real-world situations in different applications. Figure 17 shows the inference latency of RECom, TF-CPU, and TF-GPU with a batch size of 256. Besides, we compare the performance of two state-of-the-art ML compilers, XLA [15] and AStitch [78], on the GPU. Note that for dynamic-shaped graphs, these two compilers introduce significant compilation overhead. We feed the model with the same input data so that the shapes of the graph do not change, thus eliminating the compilation overhead in this experiment. Compilers like TVM [5] and BladeDISC [77] are not compared as they fail to support the compilation of typical embedding columns.

Results show that with different numbers of embedding columns, RECom achieves 14.01× to 116.23× speedups over TF-GPU. Although XLA and AStitch can bring 1.07× and 1.13× over TF-GPU on average, their improvements are much smaller than RECom. On the one hand, even with fixed input shapes, their fusion granularity is limited by the operators with dynamic output shapes. Therefore, they still generate 10,041 and 8,053 kernels for the models with 1,000 columns and cause significant non-computation overhead, which is similar to the analysis shown in Section 7.4. On the other hand, both XLA and AStitch process the embedding columns sequentially, while the workload of each column is small-scale. Therefore, they share similar wave number distribution in Figure 15 (b) and leave the GPU resource under-utilized. For TF-CPU, RECom achieves comparable performance even when the number of embedding columns is only 10. As the number of embedding columns increases, RECom achieves 1.52× to 3.07× speedups over TF-CPU. More results with various batch sizes can be found in Section E.

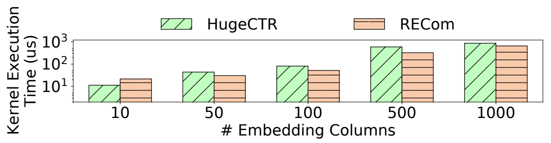


Figure 18: Comparison of embedding lookup kernel execution time with HugeCTR.

CUDA streams [45] enable the concurrent execution of multiple kernels with different streams on the GPU, which can alleviate the GPU under-utilization. However, the current TensorFlow[1] runtime does not support this feature, so XLA and AStitch have to process the columns sequentially. Besides, concurrent streams cannot avoid kernel launch overheads [45] and can introduce extra scheduling overheads. To demonstrate this, we manually bind the kernels in the 1,000 columns of the microbenchmark model into different streams and observe a 17.8× performance degradation compared to the fused kernel.

Libraries like HugeCTR. As discussed in Section 2.2, ad-hoc solutions like NVIDIA HugeCTR [43] fail to be widely adopted in companies for several reasons. However, it is still meaningful to compare RECom with them for specific operations, such as the embedding lookup. HugeCTR separates the embedding lookup into two stages, which are gathering embedding vectors and reducing vectors for each sample. As its implementation of the gathering stage involves many operations like inter-GPU caching, which is unrelated to our work, we only compare the reduction stage of HugeCTR with RECom. For embedding tables with the same embedding vector dimension, HugeCTR uses a single kernel to perform their reduction. Each block within the kernel is responsible for one sample in the batch and processes the multiple features sequentially. We compare the kernel execution time of RECom (including the entire embedding lookup) and HugeCTR (including the reduction stage only) with the batch size of 256 in Figure 18. The results show that the kernel generated by RECom has a comparable performance with HugeCTR. Besides, in real usage, RECom can handle more flexible embedding column structures and fuse more operators for greater improvement.

We also study the effect of different batch sizes on HugeCTR performance and list the results in Appendix E. We observe that HugeCTR performs better when the batch size is very large. This is because HugeCTR is mainly optimized for training processes, where large batch sizes are preferred. During inference, the batch size is usually around several hundred, for which RECom achieves similar or better performance than HugeCTR.

Combination with workload schedulers. DeepRecSys [17] and Hercules [29] are efficient workload schedulers that improve the recommendation inference throughput in heterogeneous environments. Their scheduling strategies take factors like query arrival patterns, model architectures, and accelerator characteristics into account. However, they do not accelerate the execution of inference kernels, which are entirely orthogonal to RECom. To further improve RECom's CPU utilization and overall throughput, we can use DeepRecSys to dispatch partial queries to TF-CPU.

We perform an evaluation to combine RECom and DeepRecSys [17] to optimize the throughput of recommendation model inferences further. We use 32 CPU cores and one T4 GPU during the evaluation. We take the static scheduler in [17] without RECom support as the baseline. Experimental results show that DeepRecSys achieves a 2.8× throughput improvement over the baseline, as it balances the request-level and batch-level parallelism using the hill-climbing algorithm. Enabling RECom optimization on the static scheduler also results in a 3.2× improvement. However, the naïve scheduling strategy can under-utilize partial CPUs, limiting the throughput improvement. By combining RECom with DeepRecSys, we finally obtain a 4.9× improvement over the baseline.

Manually-optimized solutions. Before RECom, our team adopts a manually-optimized CPU solution with avx512 support to serve the recommendation models. This solution extracts the matched embedding columns in the models and replaces them with optimized ones, which is labor-intensive and not flexible. We evaluate its performance on supported models A, C, and D. It fails to optimize other models due to the subgraph mismatches. Experiments show that compared to TF-CPU, this solution achieves an average speedup of 1.54× with a batch size of 256, which is significantly less than RECom (2.99× for these models with the same batch size). This is because RECom can fully exploit both intra- and inter-subgraph parallelism by utilizing the powerful GPU. The detailed experimental results are listed in Appendix E. We also compare RECom with GPU solutions that manually fuse all embedding columns. For simple and small models, such as tens of embedding columns, RECom obtains similar speedup with manually-optimized GPU solutions. However, for models with more than hundreds of embedding columns, manually performing fusion optimization is impractical.

8 RELATED WORK

Deep recommendation model optimization. Deep recommendation models have been widely used in various applications [3, 8, 9, 40, 52, 79, 80] and attracted the interest of many architecture and system researchers [17, 18, 29, 39, 50, 53, 62, 66, 68]. Many works [33, 38, 39, 50, 61, 71] are proposed to optimize the training of recommendation models, but they pay little attention to the inference tasks. DeepRecSys [17] and Hercules [29] dispatch workloads to CPUs and accelerators to improve inference throughput, which are orthogonal works to RECom. Several works [2, 43, 64] designed GPU-side embedding-caching mechanisms to compensate for the gap between the CPU-side DRAM accessing and GPU processing. In contrast, we mainly focus on models that can be fitted to the GPU. Many solutions are proposed to speed up the embedding table lookup, including near-memory processing [28, 32], GPU-based [43] optimizations, and FPGA-based [21, 25, 26] solutions. However, they do not optimize the massive operators in the embedding columns. Many implementations in MLPerf [48] are proposed to accelerate the inference of the standard DLRM [40] on Criteo [27] dataset, whose embedding part is simple. However, industrial recommendation models often contain massive embedding columns with various graph structures, which is beyond their application scope. More importantly, all of the above works are ad-hoc solutions that require huge manual efforts to rebuild models, which is intolerable for many companies.

Machine learning compilers and automatic optimizers. Many works on tensor programs optimizations have been proposed in recent years [7, 16, 36, 47, 57, 59, 60, 63, 83], like machine learning compilers [5, 15, 37, 55, 67, 77, 78, 82] and frameworks for operator fusion and pipelined scheduling [46, 75, 76]. However, these optimizers fail to optimize the embedding computations in recommendation models for the following reasons. On the one hand, most existing compilers [5, 15, 37, 78] only optimize models with static shape information and fail to handle the dynamic shape characteristics of recommendation models. On the other hand, these compilers focus on the optimizations for neural networks like MLPs, CNNs [31], and Transformers [56]. They do not design specific optimizations for the embedding computations and leave the inter-embedding-column parallelism unexploited. Many works [6, 10, 11, 19, 20, 30, 49, 51, 65, 72–74, 81] are provided to generate high-performance tensor programs of GEMM-centric (dense or sparse) graphs. In contrast, we focus on the fusion of memory-intensive operators within embedding columns. BladeDISC [77] is an ML compiler that supports dynamic shape input tensors and symbolic shape optimization but does not provide any embedding-related optimization, including compiling operators whose shapes depend on the input contents. NVIDIA provides CUDA graphs [41] to reduce the overhead of kernel launch. However, programmers can only create static-shaped graphs with CUDA graphs. Our work can well merge the GPU operations of dynamic-shaped graphs.

9 CONCLUSION

In this paper, we propose RECom, the first ML compiler that aims to accelerate the expensive embedding column processing during the inference of deep recommendation models. First, we propose the inter-subgraph parallelism-oriented fusion method to generate efficient GPU codes to process massive embedding columns in parallel. Second, we recognize the shape computation problems that arise in dynamic shape scenarios and adopt an approach based on symbolic expressions to solve them. Third, we develop an embedding column optimization module to eliminate redundant computations. Experiments show that RECom outperforms state-of-the-art baselines by 6.61× and 1.91× in terms of inference latency and throughput, respectively. We believe that RECom fills a long-overlooked gap in ML compilers for deep recommendation models.

ACKNOWLEDGMENT

We sincerely thank all the anonymous reviewers and the shepherd for their insightful comments and feedback. We also thank Donglin Zhuang, Mengli Cheng, Minmin Sun, Tongxuan Liu, Bohua Chen, and Langshi Chen at Alibaba Group for their extensive suggestions. This work is supported by the National Natural Science Foundation of China (No. 62322213 and 62172419), Beijing Nova Program (20220484137), and Public Computing Cloud, Renmin University of China. This work is also supported by Alibaba Group through the Alibaba Innovative Research Program. Zaifeng Pan, Feng Zhang, Ruofan Wu, Dalin Wang, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

A DETAILS ABOUT EMBEDDING COLUMNS

In this section, we present more details about embedding columns, including the example of building embedding columns and the illustration of the massive embedding columns.

A.1 Building Embedding Columns with High-level APIs

Developers usually rely on high-level APIs to build embedding columns. We present a simple example in Listing 1 to illustrate how to use TensorFlow FeatureColumn APIs [14] to construct a model with multiple identical embedding columns. For each input feature in string type, we first hash it into a numeric tensor (Lines 6-8) and then perform an embedding lookup (Lines 9-10). The lookup operation in Lines 9-10 already contains the safety checks discussed in Section 5.1 for robustness considerations. Finally, we concatenate all the embedding column outputs into a single tensor (Lines 13-14).

Listing 1: Example of building embedding columns

```
1  import tensorflow as tf
2
3  features = {...} # placeholder mapping
4  columns = []
5  for feat_name, feat in features.items():
6    col = tf.feature_column.
7      categorical_column_with_hash_bucket(feat_name,
8        embedding_table_rows, dtype=tf.string)
9    col = tf.feature_column.embedding_column(col,
10     embedding_dim, combiner='mean')
11   columns.append(col)
12
13 embeddings = tf.feature_column.input_layer(features,
14     columns)
```

A.2 Massive Embedding Columns

Industrial recommendation models often consist of thousands of embedding columns, as there are numerous features. The feature numbers are very large due to the following reasons. First, a typical app usually has dozens of scenes, e.g., a homepage and detail pages, and different categories like music, dance, etc. Second, for each scene, there are dozens of raw features that need to record, e.g., duration, followers, comments, and so on. Third, for each of these raw features, we need to generate its statistical characteristics, including min/max/sum/avg in different periods (e.g., one day/week/month). Besides, for these statistical features, algorithm developers often use embedding to model them [4, 35] to improve recommendation accuracy. In this way, algorithm developers can build models with a large number of embedding columns.

Besides, different embedding columns often have different graph structures. For example, they can adopt different preprocess operations and index translation functions. The numbers of embedding vectors and the embedding vector dimensions can also vary among columns. The diversity of embedding columns and their combinations makes it difficult to pre-build libraries to serve all conditions. Therefore, a compiler-based optimization approach is urgently needed.

B ILLUSTRATION OF EMBEDDING COLUMN IDENTIFICATION PROCEDURE

In this section, we illustrate the detailed procedure of embedding column identification discussed in Section 3.1.2.

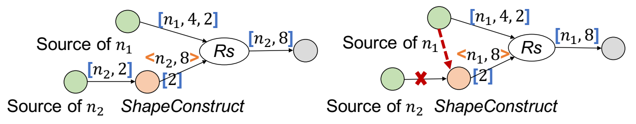
Algorithm 1 Embedding Column Subgraphs Identification

```
Input: graph $G$
Output: embedding column subgraphs
1:  count the preceding embedding tables of each operator in $G$
2:  for each $T$ in embedding tables do
3:      initialize subgraph $sg \leftarrow \{T\}$ and queue $q \leftarrow \{T\}$
4:      while $q$ is not empty do
5:          dequeue $n$ from $q$
6:          for each $m$ in $n$'s succeeding operators do
7:              if #preceding embedding tables of $m \leq 1$ then
8:                  insert $m$ into $sg$ and enqueue $m$ into $q$
9:      do
10:         for each $n$ in the operators of $sg$ do
11:             for each $m$ in $n$'s preceding operators do
12:                 insert $m$ into $sg$ if it's not in $sg$
13:     while $sg$ has been updated
14:     output $sg$
```

Algorithm 1 shows the process of identifying embedding column subgraphs. It first calculates the number of the preceding embedding tables of each operator in the graph (Line 1), and the results are illustrated in Figure 3 as the numbers beside each circle. Then, starting from each embedding table, Algorithm 1 performs a breadth-first search until finding operators with more than one preceding embedding table and inserts the visited operators into the current embedding column subgraph (Lines 3 to 8). Finally, the algorithm iteratively adds the preceding operators of all operators in the current subgraph until no more operators are added (Lines 9 to 13). This step adds the operators with no preceding embedding table to the corresponding embedding column subgraph. The final partitioned subgraphs are illustrated in Figure 3.

C SHAPE-ONLY TENSOR COMPUTATION ERASING

This section proposes shape-only tensor computation erasing, which is a minor optimization in the shape computation simplification module of RECom.



(a) A subgraph after reconstruction. n_1 and n_2 have been united.

(b) RECom replaces the n_2 input by n_1.

Figure 19: Example of shape-only tensor computation erasing.

Shape-only tensor computation refers to the operation whose outputs are consumed only by *ShapeConstruct* operators. This means that the only function of these operators is to generate symbols for later operators, and the tensor contents of their outputs are irrelevant for subsequent computations. When RECom discovers a *shape-only tensor computation*, it identifies the shape symbols generated by this operator and checks all their union symbols from other operators. If there is a symbol from a non-shape-only-tensor-computation operator, RECom then replaces the original operator with this operator to eliminate unnecessary computation. Figure 19 (a) shows a subgraph after *shape-only tensor computation erasing*. The symbols n_1 and n_2 are equal due to the input constraints. The source of n_2 in Figure 19 (a) has only one output, a *ShapeConstruct* operator, so it is regarded as a *shape-only tensor computation* operator. RECom detects this situation and replaces the input of *ShapeConstruct* from the source of n_2 to n_1.

D RANGE-BASED ELIMINATION OF REDUNDANT LOOKUP INDEX REMOVALS

In this section, we present the detailed approach to detecting and eliminating redundant lookup index removals.

We systematically summarize the four most frequent patterns of index operations in the numerical preprocess, index translation, and safety guarantees of embedding columns. Note that the lookup indices are often represented as a sparse tensor because samples within a batch can have varied item numbers or even be empty for the same feature field. The *sp_values* refer to the non-absent lookup indices in the sparse tensor, while the *sp_indices* refer to the corresponding indices for the elements in *sp_values*. As shown in Figure 20 (a), the *map* pattern maps each element of the lookup indices to a new index value. The value range of the lookup indices also changes based on the mapping rule. Figure 20 (a) uses a *Bucketize* operator for illustration. After the *Bucketize*, each lookup index is mapped to a value within $[0, 4)$. The range of propagated indices also changes from $(-\infty, +\infty)$ to $[0, 4)$. The *keep* pattern, shown in Figure 20 (b), does not change the value range of lookup indices, such as *reshape* operator. The *replace* pattern, shown in Figure 20 (c), replaces the lookup indices that do not meet the given condition. In Figure 20 (c), all indices that are not ≥ 0 are replaced by 0, for which the index value range after propagation is changed from $(-\infty, 4)$ to $[0, 4)$. The *remove* pattern, shown in Figure 20 (d), removes the indices that do not meet the specified condition from the sparse tensor. In Figure 20 (d), the elements in *sp_values* that are not ≥ 0 are removed, and the corresponding indices tuple in *sp_indices* are also removed. The propagated value range is reduced to $[0, 4)$.

By identifying the value range of each index via propagation according to the above patterns, RECom is able to remove redundant index operations. For example, if we have already identified that the input range in Figure 20 (d) is $[0, 4)$, the following *remove* computation is unnecessary and can be removed. The lookup index removal in embedding columns is usually through a subgraph of *remove* or *replace*. Therefore, if RECom verifies that the input lookup indices are all within the boundaries, then the following lookup index removal is regarded as unnecessary and will be eliminated by RECom.

In addition, RECom also merges successive *remove* or *replace* subgraphs into one to simplify the graph and reduce the computation. For example, suppose a subgraph aiming at removing values ≥ 0 is followed by a subgraph aiming at removing values < 4. In this situation, RECom constructs a new subgraph whose function is to remove values ≥ 0 or < 4 and uses this to replace the original two subgraphs.

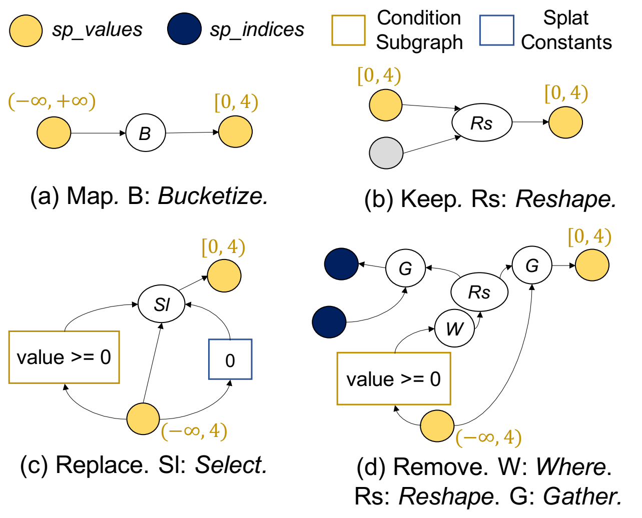


Figure 20: Common patterns of embedding lookup index operations. The intervals close to the *sp_values* indicate their value ranges.

E DETAILED EXPERIMENTAL RESULTS

In this section, we show the detailed results of experiments performed in Section 7. In Section 7.6, we present the performance comparison with prior works, including XLA [15], AStitch [78], HugeCTR [43], and the manually-optimized CPU solution. We only use a batch size of 256 in that section, as the batch size in real business is often several hundred. We present the experimental results with various batch sizes in Tables 3, 4, and 5.

For XLA [15] and AStitch [78], we observe that their inference latency changes are little as the batch size increases, which is similar to TF-GPU. This is because the wave number of most GPU kernels launched by them is far less than 1, as the analysis in Section 7.4 shows. Hence, launching more blocks for larger batch sizes does not affect the latency significantly. Besides, the non-computation overhead introduced by kernel launches, kernel synchronizations, and operator scheduling is also insensitive to the batch size.

For HugeCTR [43], we only compare the kernel execution time of the lookup procedure, as HugeCTR does not support the other computations in embedding columns (e.g., *Bucketize* in index translation). As Table 4 shows, at small batch sizes, RECom outperforms HugeCTR significantly. In contrast, at large batch sizes (1,024 and 2,048), HugeCTR achieves better performance than RECom. The reason is that HugeCTR is mainly optimized for training scenarios in which large batch sizes are often used. HugeCTR maps the lookup of one sample on all embedding tables into a thread block, which requires large batch sizes to obtain good parallelism. In the inference scenario, the batch size distribution follows the log-normal distribution [17], and most of the batch sizes are around several

Table 3: Inference latency (ms) of RECom and TF baselines as well as XLA and AStitch with microbenchmarks.

# Columns	Batch Size	TF-CPU	TF-GPU	XLA	AStitch	RECom
10	32	0.46	3.96	3.70	3.66	0.16
	64	0.54	3.95	3.76	3.61	0.16
	128	0.19	3.97	3.76	3.55	0.23
	256	0.23	4.04	3.71	3.58	0.25
	512	0.28	4.03	3.70	3.66	0.27
	1024	0.33	4.10	3.71	4.26	0.34
	2048	0.54	4.12	3.66	3.75	0.42
50	32	2.47	18.53	17.57	17.69	0.28
	64	2.83	18.51	17.42	17.24	0.28
	128	0.85	18.50	17.52	17.04	0.29
	256	0.74	18.60	18.05	17.60	0.35
	512	0.77	18.65	18.13	20.26	0.40
	1024	0.86	18.94	17.57	17.91	0.50
	2048	1.37	19.34	17.58	18.25	0.79
100	32	5.01	36.97	34.87	31.56	0.37
	64	5.63	36.89	35.63	31.72	0.37
	128	1.49	36.97	35.85	31.82	0.43
	256	1.19	37.20	35.63	31.85	0.44
	512	1.27	37.43	36.27	33.34	0.53
	1024	1.52	38.08	35.38	33.31	0.77
	2048	2.38	38.82	34.46	34.34	1.18
500	32	29.16	185.38	182.64	164.62	1.15
	64	34.59	185.59	176.35	158.49	1.21
	128	5.82	186.15	183.48	161.08	1.32
	256	5.39	187.24	181.60	165.44	1.58
	512	5.93	188.93	182.72	165.54	2.12
	1024	7.30	192.13	182.84	170.53	3.26
	2048	12.03	195.88	183.57	179.15	5.81
1000	32	82.18	378.77	372.23	327.33	2.15
	64	62.04	376.39	375.90	323.94	2.26
	128	12.81	376.96	372.84	320.12	2.53
	256	13.42	379.11	376.44	321.15	3.11
	512	14.60	382.01	373.44	329.03	4.22
	1024	16.62	388.51	373.97	338.32	6.49
	2048	32.43	396.08	371.72	346.96	11.70

Table 4: Lookup kernel execution time (μs) of HugeCTR and RECom with microbenchmarks.

# Columns	Solution	Batch Size						
		32	64	128	256	512	1024	2048
10	HugeCTR	11.1	11.1	11.2	11.2	12.9	13.4	15.9
	RECom	10.9	10.7	15.0	22.3	37.5	67.2	129.4
50	HugeCTR	43.5	43.5	43.6	43.7	51.0	66.6	92.1
	RECom	13.9	13.7	19.0	30.4	52.5	99.7	201.6
100	HugeCTR	78.3	78.3	78.4	81.3	103.9	128.0	194.4
	RECom	19.5	18.4	28.6	48.9	94.4	183.6	355.0
500	HugeCTR	377.8	377.9	446.6	476.6	526.3	629.9	966.5
	RECom	111.3	102.8	173.8	308.5	597.8	1176.6	2364.9
1000	HugeCTR	720.6	828.4	859.5	909.4	1001.3	1261.9	1989.7
	RECom	146.8	143.4	285.4	563.5	1128.8	2232.0	4098.2

hundred. As Table 4 shows, RECom achieves close or better performance for batch sizes ≤ 512. Besides, note that we use the whole lookup time for RECom while only using the reduction time for HugeCTR, which is a little bit unfair for RECom.

As shown in Table 5, the manually optimized CPU solution with avx512 support outperforms the TF-CPU by 2.90× on average across different batch sizes. By utilizing the GPU's high computing power and high bandwidth, RECom achieves a higher average speedup of 5.76× than this manually optimized solution.

Table 5: Inference latency (ms) of TF-CPU, manually-optimized CPU, and RECom.

Model	Solution	Batch Size						
		32	64	128	256	512	1024	2048
A	TF-CPU	117.2	100.5	76.6	43.8	62.0	99.3	184.7
	Manual-Opt	19.4	18.6	22.9	28.3	39.7	82.6	148.9
	RECom	8.2	9.9	13.7	21.2	39.4	69.4	133.1
C	TF-CPU	70.8	54.9	40.4	23.0	23.1	28.8	53.0
	Manual-Opt	10.2	9.9	12.1	14.0	17.2	21.9	38.0
	RECom	4.3	4.5	4.8	5.8	8.2	14.4	28.3
D	TF-CPU	96.6	83.7	54.5	31.2	31.6	35.3	54.8
	Manual-Opt	15.9	16.5	19.6	21.8	24.7	29.8	48.2
	RECom	7.9	8.6	9.2	10.6	12.2	19.0	38.4

F ARTIFACT APPENDIX

F.1 Abstract

The artifact contains the necessary software components to validate the main results in the RECom paper. We provide a `Dockfile` for users to build the docker image containing the basic environment used to compile RECom. After launching the docker container, users can pull the GitHub repository of RECom and then build and run the examples. We provide a run-all script to perform building the docker image, building RECom and the examples, creating models E and F in Section 7, running the benchmark scripts, and drawing the most important figures in the paper.

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

- **Run-time environment:** A Linux system with NVIDIA driver (capable of running CUDA 11.6).
- **Hardware:** An x86_64 CPU with at least 32 cores and an NVIDIA T4 GPU are recommended.
- **Execution:** Run a single script.
- **Metrics:** End-to-end latency and throughput.
- **Output:** Two figures in PDF format showing the performance results of RECom and TensorFlow baselines.
- **Experiments:** The end-to-end performance comparison of RECom and TensorFlow baselines on the open-source models E and F.
- **How much disk space required (approximately)?:** 32 GB.
- **How much time is needed to complete experiments (approximately)?:** Six hours.
- **Publicly available?:** Yes. RECom's source code is publicly available at https://github.com/AlibabaResearch/recom.
- **Code licenses (if publicly available)?:** Apache-2.0
- **Data licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.8379746

F.3 Description

F.3.1 How to access. We provide the `Dockfile` and the run-all script `run_all.sh` under the `AE` directory of RECom's repository. URL: https://github.com/AlibabaResearch/recom/tree/main/AE. We also zip the source codes into one file and publish it on Zenodo: https://doi.org/10.5281/zenodo.8379746.

F.3.2 Hardware dependencies. An x86_64 CPU with at least 32 cores and an NVIDIA T4 GPU are recommended.

F.3.3 Software dependencies. A Linux system with NVIDIA driver (capable of running CUDA 11.6).

F.3.4 Models. The run-all script includes creating the models E and F used in our evaluation.

F.4 Installation

Users just need to pull the GitHub repository of RECom and execute the run-all script. Note: if the compute capability of the GPU is not 7.5 or 8.6, please modify the environment variable of `TF_CUDA_COMPUTE_CAPABILITIES` in the `Dockerfile` correspondingly before running the script.

```
git clone \
    https://github.com/AlibabaResearch/recom \
    recom
cd recom/AE && ./run_all.sh
```

Alternatively, users can download the tar file through Zenodo to get the archived repository.

F.5 Evaluation and expected results

After finishing the execution of the run-all script, users can find two figures, `latency.pdf` and `throughput.pdf`, in their current directories. These two figures correspond to Figures 10 and 11 in the paper.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX OSDI 2016*, pages 265–283. USENIX Association, 2016.

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, 2021.

[3] Alibaba. Alibaba/DeepRec. https://github.com/alibaba/DeepRec, 2023.

[4] Qiwei Chen, Changhua Pei, Shanshan Lv, Chao Li, Junfeng Ge, and Wenwu Ou. End-to-end user behavior retrieval in click-through rateprediction model. *CoRR*, abs/2108.04468, 2021.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3393–3404, 2018.

[7] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data*, 1(1):1–31, 2023.

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

[9] Mengli Cheng, Yue Gao, Guoqiang Liu, Hongsheng Jin, and Xiaowen Zhang. Easyrec: An easy-to-use, extendable and efficient framework for building industrial recommendation systems. *CoRR*, abs/2209.12766, 2022.

[10] Stephen Chou and Saman P. Amarasinghe. Dynamic sparse tensor algebra compilation. *CoRR*, abs/2112.01394, 2021.

[11] Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 823–838. ACM, 2020.

[12] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells, editors, *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 191–198. ACM, 2016.

[13] ONNX developers. ONNX. https://github.com/onnx/onnx, 2023.

[14] Google. TensorFlow FeatureColumn APIs. https://www.tensorflow.org/api_docs/python/tf/compat/v1/feature_column, 2023.

[15] Google. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2023.

[16] Jiawei Guan, Feng Zhang, Jiesong Liu, Hsin-Hsuan Sung, Ruofan Wu, Xiaoyong Du, and Xipeng Shen. Trec: Transient redundancy elimination-based convolution. *Advances in Neural Information Processing Systems*, 35:26578–26589, 2022.

[17] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

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

[19] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman P. Amarasinghe, and Fredrik Kjolstad. Compilation of sparse array programming models. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021.

[20] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5, 2023.

[21] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 968–981. IEEE, 2020.

[22] Intel. Intel Performance Counter Monitor (Intel PCM). https://github.com/intel/pcm, 2023.

[23] Intel. OneAPI Deep Neural Network Library (oneDNN). https://github.com/oneapi-src/oneDNN, 2023.

[24] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun 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.

[25] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. Microrec: Efficient recommendation inference by hardware and data structure solutions. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[26] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. Fleetrec: Large-scale recommendation inference on hybrid GPU-FPGA clusters. In Feida Zhu, Beng Chin Ooi, and Chunyan Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 3097–3105. ACM, 2021.

[27] Kaggle. Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge, 2023.

[28] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 790–803. IEEE, 2020.

[29] Liu Ke, Udit Gupta, Mark Hempsteadis, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 141–144. IEEE, 2022.

[30] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman P. Amarasinghe. Tensor algebra compilation with workspaces. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 180–192. IEEE, 2019.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[32] Youngeun Kwon, Yunjae Lee, and Minsoo 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.

[33] Youngeun Kwon and Minsoo Rhu. Training personalized recommendation systems from (GPU) scratch: look forward not backwards. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 860–873. ACM, 2022.

[34] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[35] Zeyu Li, Wei Cheng, Yang Chen, Haifeng Chen, and Wei Wang. Interpretable click-through rate prediction through hierarchical attention. In James Caverlee, Xia (Ben) Hu, Mounia Lalmas, and Wei Wang, editors, *WSDM '20: The Thirteenth ACM International Conference on Web Search and Data Mining, Houston, TX, USA, February 3-7, 2020*, pages 313–321. ACM, 2020.

[36] Jiesong Liu, Feng Zhang, Jiawei Guan, Hsin-Hsuan Sung, Xiaoguang Guo, Xiaoyong Du, and Xipeng Shen. Space-efficient trec for enabling deep learning on microcontrollers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 644–659, 2023.

[37] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 881–897. USENIX Association, 2020.

[38] Meta. Pytorch domain library for recommendation systems. https://github.com/pytorch/torchrec, 2023.

[39] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 993–1011. ACM, 2022.

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

[41] NVIDIA. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[42] NVIDIA. Kernel Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide, 2023.

[43] NVIDIA. NVIDIA-Merlin/HugeCTR. https://github.com/NVIDIA-Merlin/HugeCTR, 2023.

[44] NVIDIA. NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface, 2023.

[45] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2023.

[46] Chanyoung Oh, Zhen Zheng, Xipeng Shen, Jidong Zhai, and Youngmin Yi. Gopipe: A granularity-oblivious programming framework for pipelined stencil executions on GPU. In Vivek Sarkar and Hyesoon Kim, editors, *PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, pages 43–54. ACM, 2020.

[47] Zaifeng Pan, Feng Zhang, Hourun Li, Chenyang Zhang, Xiaoyong Du, and Dong Deng. G-SLIDE: A GPU-Based Sub-Linear Deep Learning Engine via LSH Sparsification. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3015–3027, 2021.

[48] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 446–459. IEEE, 2020.

[49] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman P. Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. ACM Program. Lang.*, 4(OOPSLA):158:1–158:30, 2020.

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

[51] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.

[52] Leixian Shen, Enya Shen, Zhiwei Tai, Yihao Xu, Jiaxiang Dong, and Jianmin Wang. Visual data analysis with task-based recommendations. *Data Science and Engineering*, 7(4):354–369, 2022.

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

[54] SymEngine. SymEngine: a fast C++ symbolic manipulation library. https://github.com/symengine/symengine, 2023.

[55] Han Vanholder. Efficient inference with tensorrt. In *GPU Technology Conference*, volume 1, page 2, 2016.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[57] Haojie Wang, Jidong Zhai, Mingyu Gao, Feng Zhang, Tuowei Wang, Zixuan Ma, Shizhi Tang, Liyan Zheng, Wen Wang, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Optimizing dnns with partially equivalent transformations and automated corrections. *IEEE Transactions on Computers*, pages 1–14, 2023.

[58] Qi Wang, Zhihui Ji, Huasheng Liu, and Binqiang Zhao. Deep bayesian multi-target learning for recommender systems. *CoRR*, abs/1902.09154, 2019.

[59] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, pages 515–531, 2021.

[60] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. {MGG}: Accelerating graph neural networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} pipelining on {Multi-GPU} platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, 2023.

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

[62] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 717–729. ACM, 2021.

[63] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*, pages 1807–1816, 2022.

[64] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: an efficient gpu embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 402–416, 2022.

[65] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022.* mlsys.org, 2022.

[66] Zhiqiang Xu, Dong Li, Weijie Zhao, Xing Shen, Tianbo Huang, Xiaoyun Li, and Ping Li. Agile and accurate CTR prediction model training for massive-scale online advertising systems. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2404–2409. ACM, 2021.

[67] Xiaodong Yi, Shiwei Zhang, Lansong Diao, Chuan Wu, Zhen Zheng, Shiqing Fan, Siyu Wang, Jun Yang, and Wei Lin. Optimizing DNN compilation for distributed training with joint OP and tensor fusion. *IEEE Trans. Parallel Distributed Syst.*, 33(12):4694–4706, 2022.

[68] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: an fpga-accelerated embedding-based retrieval system. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 841–856. USENIX Association, 2022.

[69] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. A study of single and multi-device synchronization methods in nvidia gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 483–493. IEEE, 2020.

[70] Yuanxing Zhang, Langshi Chen, Siran Yang, Man Yuan, Huimin Yi, Jie Zhang, Jiamang Wang, Jianbo Dong, Yunlong Xu, Yue Song, Yong Li, Di Zhang, Wei Lin, Lin Qu, and Bo Zheng. PICASSO: unleashing the potential of gpu-centric training for wide-and-deep recommender systems. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 3453–3466. IEEE, 2022.

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

[72] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022.* mlsys.org, 2022.

[73] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 863–879. USENIX Association, 2020.

[74] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 859–873. ACM, 2020.

[75] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on GPU. In Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez, editors, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 587–599. ACM, 2017.

[76] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Hiwaylib: A software framework for enabling high performance communications for heterogeneous pipeline computations. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 153–166. ACM, 2019.

[77] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proc. ACM Manag. Data*, 1(3), nov 2023.

[78] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 359–373. ACM, 2022.

[79] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 5941–5948. AAAI Press, 2019.

[80] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1059–1068, 2018.

[81] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: fast and efficient tensor compilation for deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 233–248. USENIX Association, 2022.

[82] Kai Zhu, Wenyi Zhao, Zhen Zheng, Tianyou Guo, Pengzhan Zhao, Junjie Bai, Jun Yang, Xiaoyong Liu, Lansong Diao, and Wei Lin. DISC: A dynamic shape compiler for machine learning workloads. In Eiko Yoneki and Paul Patras, editors, *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*, pages 89–95. ACM, 2021.

[83] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems*, 4:316–336, 2022.