RecFlex: Enabling Feature Heterogeneity-Aware Optimization for Deep Recommendation Models with Flexible Schedules

Zaifeng Pan*§, Zhen Zheng†, Feng Zhang*, Bing Xie†, Ruofan Wu*§, Shaden Smith†,
Chuanjie Liu†, Olatunji Ruwase†, Xiaoyong Du*, Yufei Ding‡
*Renmin University of China, †Microsoft, ‡University of California, San Diego
{panzaifeng,fengzhang,ruofanwu,duyong}@ruc.edu.cn
{zhengzhen,bingxie,shadensmith,chuanli,olruwase}@microsoft.com
yufeiding@ucsd.edu

***Abstract*—Industrial recommendation models typically involve numerous feature fields. The embedding computation workloads are heterogeneous across these fields, thus requiring varied optimal code schedules. While existing solutions apply basic fusion optimization for embedding operations, they inefficiently treat all feature fields with identical schedules, leading to suboptimal performance. In this paper, we introduce RecFlex, which generates fused kernels with distinct schedules for different feature fields. RecFlex employs the *interference-aware schedule tuner* to tune schedules and the *heterogeneous schedule fusion compiler* to generate fused kernels, addressing two major challenges. To determine optimal schedules of different feature fields within the fused kernel, RecFlex proposes a two-stage interference-simulated tuning strategy. To handle dynamic workloads that challenge tuning and fusion, RecFlex combines compile-time schedule tuning with runtime kernel thread mapping. RecFlex surpasses state-of-the-art libraries and compilers, achieving average speedups of 2.64×, 20.77×, and 11.31× over TorchRec, HugeCTR, and RECom, respectively. RecFlex is publicly available at https://github.com/PanZaifeng/RecFlex.**

***Index Terms*—recommender system, machine learning compiler**

I. Introduction

In recent years, deep recommendation models have been extensively utilized across enterprises and business scenarios, such as video rankings at Google Youtube [1], [2], online advertising applications at Meta [3]–[5], and e-commerce at Alibaba [6]–[10]. Unlike other prevalent deep neural networks (DNNs), a typical recommendation model exhibits a distinctive structure featuring two key parts: the embedding layers and the DNN layers. Specifically, the embedding layer transforms the inputs of different *features*[1] (e.g., user IDs and product IDs) into representative embedding vectors by performing embedding table lookup and pooling operations. To achieve high model quality, developers have incorporated hundreds to thousands of features [11]–[13] into their models in production. These features introduce numerous memory-intensive embedding operations, making the execution of embedding layers quite time-consuming [12], [14].

Traditional machine learning (ML) frameworks [15], [16] execute embedding operations (including embedding lookup and pooling) separately for each feature, resulting in low GPU utilization and significant GPU kernel launch overhead [12], [14]. To tackle this performance problem, existing solutions [12], [14], [17], [18] fuse the embedding operations through manually written libraries or compilation optimization. These works treat all features equally with identical code schedules[2] in the fused kernel. However, we find that the numerous features in an industrial recommendation model are heterogeneous, i.e., the computation workloads vary significantly among features (detailed in Section II-A). Existing solutions that apply a single code schedule for heterogeneous features within a model lead to poor performance for most features.

Given the limitations of existing solutions, there is a potential to further optimize the embedding operations by generating distinct schedules for different features according to their workload characteristics. However, enabling this optimization poses two major challenges. First, it is challenging to find the optimal schedules of all features in the final fused kernel of the embedding operations. Directly combining the best schedule of each feature based on its separate latency will fail to achieve the overall best performance due to interference among schedules in the fused kernel. The interference includes occupancy constraint, i.e., the maximum number of active warps per streaming multiprocessor (SM), and resource contention. Enumerating the schedule combinations of all features is also impractical, as the search space is too large. Second, the dynamic computation workloads of recommendation models make both kernel fusion and schedule tuning challenging. Existing code schedule generators and tuners [21]–[26] rely on static workloads to guide thread mapping of the generated kernels and optimal schedule identification.

§Work was done when Zaifeng and Rufan interned at Microsoft, advised by Zhen.

[1]We use *feature* to represent *feature field* in this paper for abbreviation.

[2]In the field of ML compilers and code generation, schedule [19], [20] refers to how the code is organized to map to hardware, like the tiling approach, thread mapping method, loop orders, etc.

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

To this end, we propose RecFlex, a system that optimizes the time-consuming embedding operations of recommendation models by enabling feature heterogeneity-aware optimization with flexible schedules. RecFlex comprises two key components, the *interference-aware schedule tuner* and the *heterogeneous schedule fusion compiler*. The schedule tuner is responsible for efficiently identifying the optimal schedules for all features, while the fusion compiler generates the fused GPU kernel based on the per-feature schedules.

To address the first challenge of optimal schedule identification, RecFlex's schedule tuner adopts a two-stage interference-simulated strategy that accounts for inter-feature interference. It first tunes the optimal per-feature schedule with different occupancy constraint values in the *local* stage. This stage involves controlling occupancy values explicitly to eliminate the per-feature schedule's dependency on the fused kernel occupancy and padding thread blocks to simulate inter-feature resource contention. Subsequently, in the *global* stage, it further tunes the overall optimal occupancy value. To solve the challenge of dynamic workloads, we propose a combined approach. During compilation, we leverage the recent distribution of historical inputs to tune generally optimal schedules and update them periodically. At runtime, we analyze the input workload on the host side to determine thread mapping, thus averting GPU workload imbalances or resource wastage.

We integrate the fused kernel generated by RecFlex with PyTorch [15] framework and evaluate RecFlex on recommendation models and datasets synthesized based on our observation of production models. Experimental results show that compared to TorchRec [17], HugeCTR [18], and RECom [12], the fused kernels generated by RecFlex achieve average speedups of 2.64×, 20.77×, and 11.31×, respectively. We summarize our contributions as follows:

- We reveal the feature heterogeneity in industrial deep recommendation models and identify the limitations of existing solutions.
- We propose RecFlex, a recommendation model optimization system that enables feature heterogeneity-aware optimization in the fused GPU kernels with flexible schedules. To our knowledge, this is the first work to discuss schedule tuning considering inter-schedule interference for horizontal fusion.
- Extensive experiments demonstrate the effectiveness of our tuning strategy and showcase that RecFlex achieves significant improvements over state-of-the-art baselines.

II. Background and Motivation

A. Feature Heterogeneity of Recommendation Models

Deep recommendation models and embedding operations. As shown in Figure 1, a typical deep recommendation model's embedding layers contain a set of embedding tables, which transform the input from various feature fields into representative embedding vectors through embedding operations. The embedding vectors are then concatenated to be fed into the subsequent DNNs to predict the output value (e.g., the click-through rate). The dotted rectangle in Figure 1 illustrates the

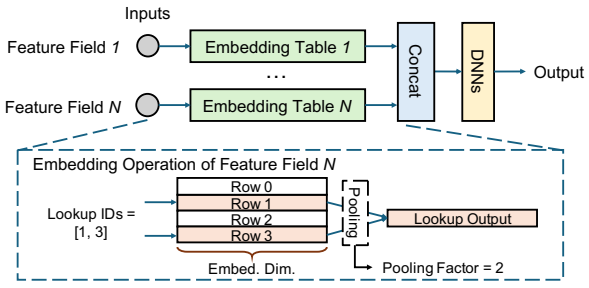
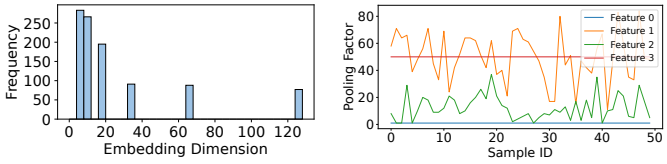


Fig. 1. A typical deep recommendation model architecture with N features.



(a) Embed. dimension distribution. (b) Pooling factor distribution.

Fig. 2. Embedding dimension and input workloads can vary significantly among features.

embedding operations for feature N. An input sample of the operation consists of single or multiple lookup IDs, which are used to retrieve corresponding rows in the embedding table. Then, the retrieved embedding vectors of the same sample will be applied by a pooling operation (i.e., element-wise reduction) to get the final lookup output. Production recommendation models often contain thousands of features, and the corresponding embedding operations can account for most of the end-to-end execution time [11], [12].

Heterogeneity across features. By profiling the production models deployed at real businesses, we observe that the embedding table characteristics and input workloads vary significantly among features. For a specific feature, we denote the number of retrieved embedding vectors in a particular sample as *pooling factor* [13], and the row vector dimension of its embedding table as *embedding dimension*. Figure 2(a) reports the embedding dimension distribution of a recommendation model, which ranges from single digits to hundreds. Figure 2(b) presents the pooling factors of four features in 50 samples. Due to variations in embedding dimension and pooling factors, each feature has distinct memory access and computation patterns. We use *feature heterogeneity* to represent this phenomenon in the paper.

B. Limitation of Existing Solutions and New Opportunity

To avoid the inefficient separate execution of embedding operations for different features, existing works [12], [14], [17], [18] propose fusing all embedding operations into a single GPU kernel through a library or compiler approach. For example, RECom [12], an optimizing compiler for recommendation model inference, achieves an 11.20× speedup on production models after enabling cross-embedding fusion.

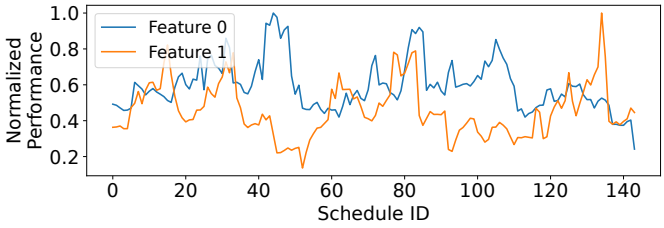


Fig. 3. Normalized performance of different schedules on particular features with their distinct workloads. The embedding dimensions for both features are 32. Pooling factors for feature 0 follow a normal distribution $N(50, 10^2)$ with a 0.3 coverage [13], while feature 1 has fixed pooling factors of 50.

However, we observe a key limitation of these works: they treat all features equally within their fused kernels. NVIDIA HugeCTR library [18] only supports creating an embedding layer with the same embedding dimension for all features. RECom [12] evenly distribute the embedding operations of different features to individual GPU blocks. TorchRec [17] selects the pre-compiled fused kernels based on the maximum embedding dimension among all tables. They all overlook the significant feature heterogeneity common in real businesses.

Opportunity: apply distinct schedules for different features in the fused kernel. To quickly verify the feasibility of this idea, we conduct a microbenchmark to evaluate the performance of different schedules on two particular features whose computation workloads are different, and show the results in Figure 3. We observe that, for a specific feature, different schedules can have a performance gap of up to 86.4%. Additionally, the optimal schedules of these two features are not the same due to feature heterogeneity. Therefore, it has great potential to improve the fused kernel performance by enabling distinct schedule optimization to handle feature heterogeneity.

C. Challenges

In this section, we mainly discuss the challenges of enabling heterogeneous schedule optimization and the problems of several straw-man solutions.

Challenge I: determining optimal schedules in the fused kernel. Existing code schedule generation works [21]–[25] are designed for conventional DNN models, where the execution of multiple operators is performed sequentially so that they can tune each schedule (corresponding to a fusion group or an operator) one by one separately. In contrast, for the embedding operations of recommendation models, we have to tune the schedules for numerous features in the fused kernel and consider inter-feature interference.

Straw-man solution 1: tune separately and combine. A straightforward way is to tune the schedules separately by generating and measuring non-fused kernels for each feature, and then combine these schedules together. However, lower separate latencies do not always indicate a lower latency of the fused kernel due to inter-feature interference. Each feature's selected schedule might utilize more resources like SMs and involve more memory accesses. It can introduce intensive resource contention among different features in the fused kernel, causing overall performance degradation. Moreover, the selected schedule might constrain the overall kernel occupancy due to high shared memory and register usage. This can significantly impact other features if their corresponding schedules rely on a high occupancy to increase concurrency and hide instruction latency.

Straw-man solution 2: tune holistically. Another way is to regard the fused kernel containing various schedules as a holistic entity and then tune it by enumerating all schedule combinations. However, this method is intractable as it requires exponential compilation and measurement time. For example, assume we have a model with $F = 100$ features, and for each feature, we have $N = 4$ schedule candidates for tuning. Then, we have $N^F = 4^{100} \approx 10^{60}$ combinations, and it is impossible to compare such a large number of kernels.

Challenge II: handling dynamic workloads. As shown in Figure 2(b), the pooling factors of input tensors of a model can vary significantly among samples. This causes the computation workloads to be only known at runtime. The varied batch sizes and the absence of features also contribute to the dynamics. Dynamic workloads make it difficult to determine the thread mapping of each schedule in the fused kernel, i.e., use which thread groups to process each of the features. Existing libraries [17], [18] use a uniform schedule so that each thread group can identify its responsible feature in easy ways, like dividing the group ID by a constant number. These simple methods no longer work well when using heterogeneous schedules, as the required thread group number varies across features. Furthermore, dynamic workloads challenge the tuning processes of existing schedule tuners [21], [22], [25], which require static workload information to measure the schedule performance.

Straw-man solution to thread mapping: static mapping based on the maximum/average historical workloads. A naïve idea is to allocate fixed thread groups to each feature based on its maximum or average historical workloads and then adjust the per-thread workload adaptively. In this way, each thread group can identify the feature to process at compile-time, thus avoiding the complexity of runtime thread mapping. The problem with this solution is that the workload varies significantly among samples and batches. For example, the standard deviation of pooling factors can be up to hundreds. In this case, if we aggressively allocate the thread groups based on the maximum workload, most of the threads can be idle during the computation. If we use the average workload instead, performance will also be affected when large workloads arrive, leading to a workload imbalance among GPU thread groups.

Straw-man solution to schedule tuning: runtime schedule selection. To ensure the operator performance with arbitrary workloads, some dynamic shape compilers [24] adopt to adaptively select the most suitable schedule for the separated operators in DNN layers based on input shapes at runtime. However, when applied to the embedding layers, this method requires preparing N^F combinations of schedules for runtime selection, which is infeasible. Another alternative is

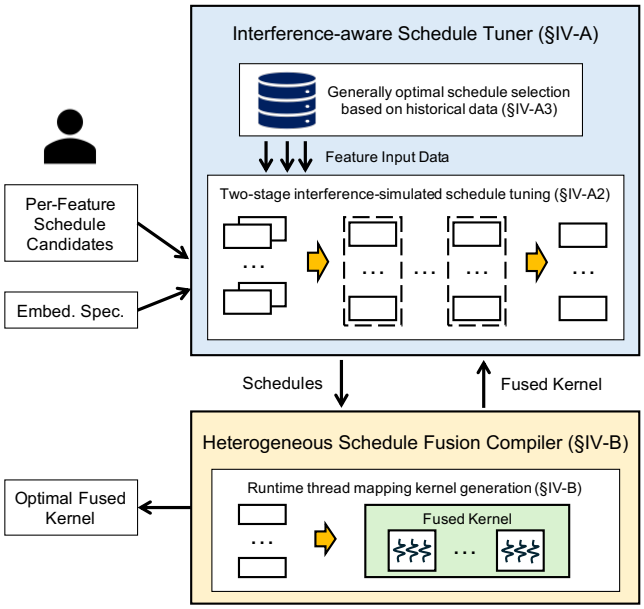


Fig. 4. RecFlex overview.

to accommodate all the schedules into the fused kernel and use internal branches for each feature to facilitate runtime selection. While this approach reduces the number of compiled kernels, overall occupancy is constrained by the most resource-intensive schedules, potentially decreasing performance. Note that no matter whether they will be executed at runtime, all branches of a GPU kernel are compiled and impact the overall occupancy. Besides, this alternative exacerbates the challenge of thread mapping, as different schedules can demand distinct thread mapping schemes.

In summary, the straw-man solutions of holistic tuning and runtime schedule selection are intractable with exponential time complexity, while the separate-combine tuning and static thread mapping can lead to sub-optimal performance. We present the performance analysis of the latter two solutions in Section VI-D.

III. RecFlex Overview

We present the system overview of RecFlex in Figure 4, which contains the *interference-aware schedule tuner* (Section IV-A) and the *heterogeneous schedule fusion compiler* (Section IV-B). To use RecFlex to optimize the embedding operations, users are required to provide per-feature schedule candidates and embedding table specifications. Then, the schedule tuner of RecFlex will tune the schedules based on the historical input data of the models. The tuner relies on the fusion compiler to generate fused GPU kernels of the being-tuned schedules for latency measuring. Finally, the tuner figures out the optimal schedules of all features and feeds them to the compiler to output the optimal fused kernel.

Solution to challenges. RecFlex enables the feature heterogeneity-aware optimization by addressing the challenges discussed in Section II-C effectively. To determine the optimal schedules in the fused kernel, the schedule tuner of RecFlex adopts a two-stage interference-simulated schedule tuning strategy (Section IV-A2) that considers inter-feature interference and tunes the schedules in polynomial time. To solve the problems of dynamic workloads, we propose a combined approach that tunes the generally optimal schedules based on historical data at compile-time (Section IV-A3) and determines the thread mapping based on input workloads at runtime (Section IV-B).

IV. RecFlex Design

A. Interference-aware Feature Schedule Tuner

This section presents the design of RecFlex's schedule tuner. The tuner identifies the generally optimal schedules by taking inter-feature interference into account, including overall occupancy constraints and resource contention.

1) Problem definition: Suppose we have a model with F features, and for each feature $f \in [1, F]$, we have N_f schedule candidates $S^{(f)} = \{S_1^{(f)}, \cdots, S_{N_f}^{(f)}\}$. We define the selected set of schedules in the fused kernel as $\mathbf{s} = \{s^{(1)}, \cdots, s^{(F)}\}$, where $s^{(f)} \in S^{(f)}, \forall f \in [1, F]$. Then, the target of the schedule tuner is to find the set of schedules $\mathbf{s}$ in the search space $S = \{S^{(1)}, \cdots, S^{(F)}\}$, that:

$$\min_{\mathbf{s}} L(\mathbf{s}; \xi) \tag{1}$$

where ξ is the input data, and $L(\cdot)$ is the fused kernel latency with given schedules and data. Let $l_b(\cdot)$ be the execution time of GPU thread block b with the given schedules and inputs, $l_b^{(f)}(\cdot)$ be the time of block b used to process feature f (we suppose different features use separate block groups, which will be discussed in Section IV-B), and $\xi^{(f)}$ be the input data of feature f. Then, we can get an approximation:

$$L(\mathbf{s}; \xi) \approx \frac{\sum_b l_b(\mathbf{s}; \xi)}{\#SM \cdot O/W} = \frac{\sum_f \sum_b l_b^{(f)}(s^{(f)}; \xi^{(f)})}{\#SM \cdot O/W} \tag{2}$$

where $\#SM$ is the number of SMs on the GPU, O is the occupancy of the fused kernel, i.e., the maximum active warps per SM, and W is the number of warps per thread block. Figure 5 shows a GPU kernel execution example to illustrate this approximation. In the figure, we suppose the GPU has two SMs, and the maximum number of active blocks per SM is $O/W = 2$, so the number of blocks that can be executed in parallel is $\#SM \cdot O/W = 4$. Whenever a running block exits, the GPU scheduler immediately schedules a pending block to the released SM. After a block is dispatched to an SM, it runs non-preemptively until it finishes. Therefore, we can approximate the kernel latency by summing up the execution time of all blocks and then divide it by the number of parallel blocks, if the total number of blocks is large enough so that the tail effect is negligible. Based on our observation, even during online serving, the batching systems [5], [16] can dispatch a large enough batch of requests to a GPU, so we consider the approximation to be reasonable.

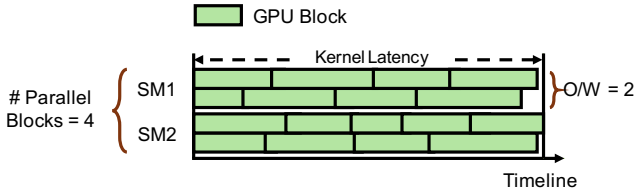


Fig. 5. An illustration example of the kernel execution on the GPU.

2) Two-stage interference-simulated schedule tuning: It is impractical to tune all schedule combinations to solve Equation 2 directly. Instead, we decompose the original problem into F sub-problems, where the objective for each sub-problem is to find the schedule $s^{(f)}$ that:

$$\min_{s^{(f)}} \left[\sum_b l_b^{(f)}(s^{(f)}; \xi^{(f)}) \right], \ f \in [1, F] \tag{3}$$

with the overall kernel occupancy O constraint. Due to the inter-feature interference, we cannot directly solve each sub-problem independently. We need to eliminate the occupancy dependency of each sub-problem on the schedules of other features and consider the implicit inter-feature interference.

Two-stage occupancy-controlled tuning. During tuning the schedules of different features in separate kernels according to Equation 3, we control the occupancy of all features explicitly to provide a uniform O for all sub-problems. This is achieved by explicitly limiting register usage and spilling overflowed registers to global memory for kernels with low occupancy, while padding shared memory usage for kernels with high occupancy. With the explicitly controlled occupancy, we introduce a *local-global* two-stage approach to tune the optimal schedules:

- *Local* stage: With given schedule candidates of each feature, we first enumerate all possible occupancy values (the count is often less than ten) based on the GPU hardware. Then, for each occupancy value, we perform the interference-simulated schedule tuning (detailed in the next part) to solve Equation 3 to get the optimal schedule of each feature.
- *Global* stage: After obtaining the optimal schedule set with each possible occupancy value, we utilize the fusion compiler of RecFlex to generate the corresponding fused kernel and measure its performance. Finally, we choose the optimal occupancy and schedules based on their corresponding kernel performance. Formally, the goal of this stage is to find the occupancy O_k that minimizes:

$$\min_{O_k} L_{O_k}(\mathbf{s}_k; \xi) \tag{4}$$

where $\mathbf{s}_k$ is the selected schedule set corresponding to occupancy O_k in the *local* stage, and $L_{O_k}(\cdot)$ is the fused kernel latency with explicitly controlled O_k.

We show this two-stage procedure in Figure 6. Suppose K is the number of possible occupancy values, so in the *local* stage, we tune the optimal schedules for all the occupancy values through O_1 to O_K. Then, we perform occupancy tuning in the *global* stage to determine the optimal occupancy value and schedule set. For the per-feature schedule tuning in the local stage with a specific occupancy, we only need to compile one kernel (detailed in the next part), so the time complexity of the *local* stage is $O(F \cdot K)$, and the complexity of the entire two stages is $O(F \cdot K + K)$. This means that we can solve the problem in polynomial time.

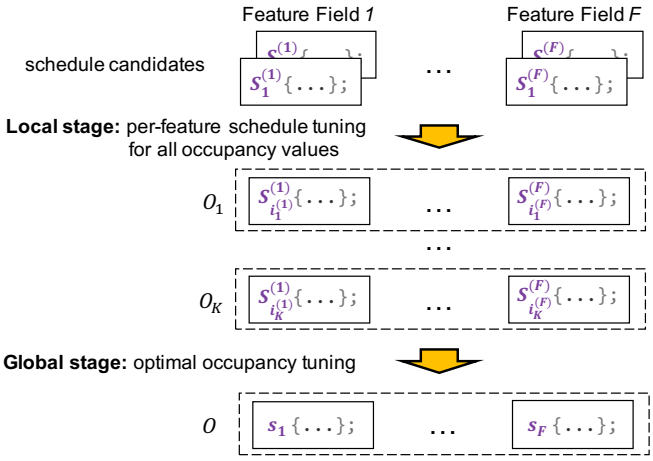


Fig. 6. The two-stage schedule tuning procedure of RecFlex. $O_1, \cdots, O_K$ are K possible occupancy values. $i_k^{(f)}$ refers to the index of optimal schedule candidate for feature f with occupancy O_k.

Interference-simulated per-feature schedule tuning. To effectively tune per-feature schedules in the *local* stage, we propose an interference-simulated approach to simulate the running environment with inter-feature interference of the fused kernel and effectively compare different schedule candidates. The insight is that although it is quite difficult to predict the block execution time $l_b^{(f)}(\cdot)$ accurately to solve Equation 3 due to inter-feature interference [27], [28], we can compare the schedules under the same simulated environment without the exact value of $l_b^{(f)}(\cdot)$.

For each feature f, we simulate the GPU SM-level and grid-level resource contention in the final fused kernel by padding GPU blocks. First, this padding strategy fills the SMs with the maximum number of allowable parallel blocks, enabling the simulation of resource contention and instruction latency hiding at the SM level. Without such padding, a single feature's input workload that fails to saturate the GPU would result in the launched blocks being dispatched to separate SMs due to the GPU's round-robin scheduling. This leads to an absence of intra-SM interference, making that changing the controlled occupancy does not affect the block execution times. Second, by incorporating redundant embedding operations within these padding blocks, we can simulate the grid-level global memory and L2 cache access patterns.

Furthermore, for effective comparison, we accommodate all schedule candidates in the same GPU kernel so that we can co-execute them in the same environment. Then, according to Equation 3, for each schedule candidate, we measure and sum

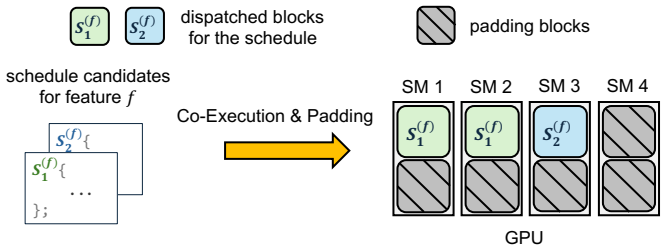


Fig. 7. Illustration of interference-simulated schedule tuning of feature f.

its corresponding block execution time and compare it with others. For the candidate with the lowest execution time sum, we regard it as the optimal one that outperforms others in the final fused kernel, which involves many resource contention.

Figure 7 shows an illustration example of the interference-simulated tuning with co-execution of schedule candidates and block padding. We rely on the schedule fusion compiler of RecFlex to generate the kernel with co-execution and padding and duplicate the input data $\xi^{(f)}$ for all fused schedules. Although the simulation cannot provide the totally same environment as in the final fused kernel, we find this approach simple yet effective for the optimal schedule tuning (see Section VI-D for the evaluation).

3) Generally optimal schedule selection based on historical data: As discussed in Section II-C, it is infeasible to perform runtime schedule selection due to the extremely large number of combinations. Meanwhile, recent works [5], [13] point out that the input data of a recommendation model follow the same distribution in a certain time period, which provides them with opportunities to predict future distribution by profiling historical data. With this insight, we measure the block execution time on the historical data in a certain period during tuning and then choose the schedules with the lowest average time. That is, we adjust the problem definition from Equation 1 to:

$$\min_{\mathbf{s}} \left[\sum_{i} L(\mathbf{s}; \xi_i) \right] \tag{5}$$

where ξ_i is the i-th batches in the sampled historical data. Besides, we re-tune the schedules periodically (e.g., several days) to handle the distribution shifts [11], [13]. In this way, the selected schedules are generally optimal in each period.

B. Heterogeneous Schedule Fusion Compiler

In this section, we present the design of RecFlex's *heterogeneous schedule fusion compiler*, which generates a fused GPU kernel (shown in Figure 8) to process different features with flexible schedules. We illustrate how the compiler addresses the challenge of dynamic workloads and discuss several design choices during codegen.

Runtime thread mapping with host-side workload analysis. As discussed in Section II-C, static thread mapping at compile-time cannot meet the requirements of the dynamic workloads, so we resort to the runtime thread mapping based on the input workloads. In this way, we can allocate an adaptive number of GPU thread groups to avoid workload imbalance and resource wastage. However, performing runtime thread mapping is a non-trivial task, as each thread group needs to identify its corresponding feature and its relative position within the groups for the same feature. Directly analyzing the inputs within the fused GPU kernels and then performing thread mapping requires the technologies of persistent threads [29], [30] or dynamic parallelism [31], which are complicated and can cause performance degradation. Instead, we utilize the host-side CPUs to determine the thread mapping and then pass the mapping information to the GPU. Besides its simplicity, we adopt this design for two reasons. First, before performing embedding operations, the input features are often applied with preprocess operations such as string split on the CPUs within the same node or from other nodes [8], [12], [32]. Therefore, we can modify these preprocess operations by adding extra workload analysis per data reading, thus hiding the computation overhead. Second, we only need to retrieve several bytes from DRAM for each thread group, which introduces negligible kernel overhead.

Feature thread mapping unit selection. We choose the GPU thread block as the basic thread mapping unit and dispatch multiple blocks to each feature. The first reason is convenience, as GPU thread blocks have separate shared memories, which is easy for management. CUDA also provides many convenient block-level intrinsic instructions, like the synchronization barrier. Second, choosing block as the basic unit is suitable for recommendation inference workloads, where the batch size of most queries is around hundreds. Note that the key idea of our design is not limited to blocks but can be extended to other thread group structures like warps or block-clusters [31].

Argument passing. There can be thousands of schedules to fuse, and each schedule can accept multiple input arguments. Therefore, we cannot directly concatenate all the arguments for the fused kernel entry, as the number of arguments for a single function in CUDA and C/C++ is limited. Instead, we pass an array of pointers on the GPU to the fused kernel, which points to the real required arguments so that the schedules can use specific indices to access their arguments.

If-else branches vs function pointer array. Using if-else branches to dispatch the processing of different features to the GPU blocks within the fused kernel can lead to thousands of integer comparisons, while using a function pointer array can directly jump to the destination function. However, we find that using function pointer arrays can actually lead to 45.0% performance degradation compared to if-else branches. This is because it leads to significant function call overhead on the GPU, while using the block-level branches can make all functions inline within the branches and only introduce negligible overhead, even with thousands of branches.

We show an illustration example of the fused kernel in Figure 8. We pass the thread mapping information in an array called `d_task_map` to the fused GPU kernel (Line 7 in the figure). In Line 9, we retrieve the thread mapping information of each block, which includes `feature_idx` that contains

```
1  __device__ void Schedule1(...) { ... }
   ...
2  __device__ void ScheduleN(...) { ... }

3  __device__ int arg_offsets[N] = { ... };
4  __device__ int schedule_map[N] = { ... };

5  __global__ void
6  __launch_bounds__(BLOCK_THREADS, MAX_REG)
7  FusedKernel(void *args, int2 *d_task_map,
               int *d_blocks_map) {
8    int2 task = d_task_map[blockIdx.x];
9    int feature_idx = task.x, rel_bidx = task.y;
10   int feature_blocks = d_blocks_map[feature_idx];
11   int schedule_id = schedule_map[feature_idx];

12   __shared__ union {
13     Schedule1SharedMemory s1;
       ...
14     ScheduleNSharedMemory sN;
15   } s;

16   if (schedule_id == 1) {
17     Schedule1(args + arg_offsets[feature_idx],
18               rel_bidx, feature_blocks, &s);
19   } else if (schedule_id == 2) {
     ...
20   } else if (schedule_id == N) {
21     ScheduleN(args + arg_offsets[feature_idx],
22               rel_bidx, feature_blocks, &s);
23   }
24 }
```

Fig. 8. An example of the fused kernel generated by RecFlex compiler.

the index of feature to process and `rel_bidx` that refers to the relative block index in the blocks corresponding to the same feature. We also retrieve `feature_blocks`, which is the total number of blocks allocated for the feature, in Line 10 by another map array. For Line 12-13, we define the shared memory usage of the fused kernel based on the maximum usage of all schedules, as the schedules use separate blocks with no shared memory overlap. For Line 16-23, we call the schedule based on the `schedule_map`, which maps the `feature_idx` to the shared schedules. Features with identical embedding dimensions and workloads can have the same optimal schedule, and sharing this schedule can reduce code size and accelerate compilation.

V. Implementation

We implement the core design of RecFlex in 1.6K lines of Python codes. We also develop several example schedule templates for the irregular embedding operations with 2K lines of CUDA and Python based on the kernels provided by TensorFlow [16], TorchRec [17], and NVIDIA Thrust [33]. The generated fused GPU kernel is wrapped and registered as a PyTorch operator [15]. Users can rely on the PyTorch API of `torch.ops.load_library` to load the compiled binary so that they can conveniently utilize the fused embedding operator in their models to accelerate the inference.

RecFlex allows users to add customized schedule templates by inheriting a schedule template class using Python. In the derived class, users need to implement the interface functions to emit the schedule code within the device function body (Line 1 in Figure 8) based on given parameters and pre-defined variables (e.g., `rel_bidx` at Line 9 in Figure 8), provide the used shared memory, and define the search space of tunable schedule-related parameters.

TABLE I
Basic statistics of evaluated models and datasets.

Model	# Features	# One-hot	# Multi-hot	Emb. Dim.
A	1,000	500	500	4-128
B	1,200	1,000	200	4-128
C	800	0	800	4-128
D	1,000	500	500	8
E	1,000	500	500	32

VI. Evaluation

A. Experimental Setup

Models and datasets. The commonly used datasets [34], [35] are too simple to be representative of current production models [11], [13], as they only have tens of features and exhibit low feature heterogeneity. Therefore, we synthesize some datasets based on our observation of production data for evaluation, using different portions of feature types and data distributions to see whether RecFlex can handle various configurations effectively. We provide a script to generate the input datasets (i.e., the embedding lookup indices) with specified pooling factor distribution and embedding table shape of each feature. Table I shows the basic statistics of these models and datasets. The details of the model and dataset configurations, as well as the data generation script, can be found in our artifact repository[3]. Models D and E share the same input dataset while their embedding dimensions are different, and they both use fixed embedding dimensions for all features to evaluate HugeCTR's [18] fused embedding operation (detailed in the next paragraph).

Baselines. We compare RecFlex with other recommender frameworks, including TensorFlow (TF) [16], TorchRec [17], HugeCTR [18]. TensorFlow [16] does not fuse the embedding operations and processes each feature sequentially, which is inefficient. Hence, we also add the baseline of RECom [12], an end-to-end optimizing compiler for TensorFlow recommendation models, which fuses all embedding operations into a single GPU kernel. TorchRec [17] is a PyTorch domain library for building recommendation models. We use the `FusedEmbeddingBagCollection` for TorchRec during the evaluation, which leverages FBGEMM [36] to enable the fused embedding operations. NVIDIA HugeCTR [18] allows users to create an embedding layer containing multiple tables with the same vector dimension so that it can concatenate these embedding tables to perform fused operations. For HugeCTR, we only evaluate its performance on models D and E, which have fixed embedding dimensions across features.

Testbed. We evaluate RecFlex and the baselines on an NVIDIA V100 GPU an A100 GPU, respectively. The Python and CUDA versions used in the evaluation are 3.8 and 11.8, respectively. For the evaluation of RecFlex and TorchRec, we

[3]https://github.com/PanZaifeng/RecFlex/tree/main/data_synthesis

use PyTorch with the version 2.2.0. The TorchRec version is 0.6.0 with FBGEMM 0.6.0. For TensorFlow and RECom, we use TensorFlow 2.6.2, which is the version supported by RECom [12]. For HugeCTR, we use its official image with version 23.04.

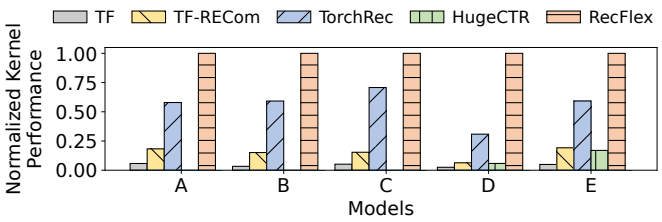
Evaluation method. We evaluate the inference latency of recommendation models, but there is no fundamental reason limiting RecFlex from optimizing the training process, except the manual efforts to support more operators. We randomly sample 128 batches for each evaluation model from its corresponding dataset. Then, we read each batch of input data from the disk and perform the preprocess in advance. We then measure the model execution time, accumulate the execution time of these batches, and normalize it to the most performant framework. Note that in production, the preprocess is executed with many CPU threads by using highly optimized libraries, and the optimization of the preprocessing is not the focus of our paper. Other works [5], [14], [17], [18], [37] also adopt a similar setting. We use NVIDIA Nsight Systems to measure the kernel execution time accurately.

B. Kernel Performance

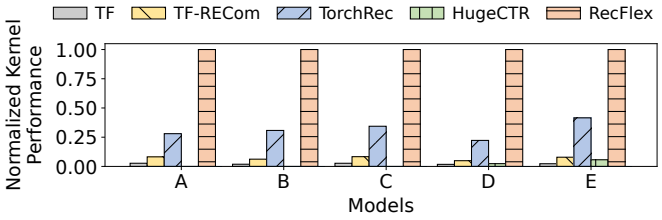
We first evaluate the performance of the fused embedding kernel. Figures 9(a) and (b) reports the normalized performance of RecFlex and the baselines on a V100 GPU and an A100 GPU, respectively. For the TensorFlow baseline, which launches separate GPU kernels for different features, we summarize the execution time of all the embedding kernels for it. For the HugeCTR baseline, we only show its performance on models D and E, whose embedding dimensions are the same across feature. Besides, we find that HugeCTR's embedding layer involves many other operations related to its GPU-side embedding cache mechanism, such as cache update operations. These operations cannot be avoided even though we place the entire embedding table on the GPU memory. Hence, to compare HugeCTR with others fairly, we ignore its cache-related operations and only measure the execution time of its fused pooling operation kernels. The results show that across these models and datasets, RecFlex achieves average speedups of 35.40×, 11.31×, 20.77×, and 2.64× over TensorFlow, RECom, HugeCTR, and TorchRec on the two GPU platforms.

The performance of TensorFlow is very poor, as it executes all embedding operations sequentially. RECom accelerates the execution of TensorFlow significantly by automatically fusing the embedding operations to improve GPU utilization, but its performance is still limited by inefficient schedules and static thread mapping. TorchRec adopts a fine-grained sample-warp parallelism mapping and shows the best performance among the baselines, but it still has the problem of overlooking the feature heterogeneity within a recommendation model.

We observe that the performance of HugeCTR is worse than RECom and TorchRec, even though HugeCTR is a vendor-provided library and requires the same embedding dimension across features. The reason is that HugeCTR uses a coarse-grained sample-block parallelism mapping and processes all features sequentially within a block, which relies on large



(a) Performance on a V100 GPU.



(b) Performance on an A100 GPU.

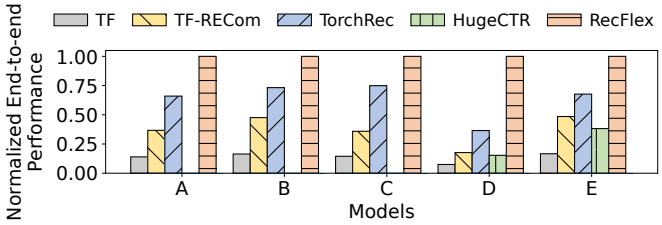
Fig. 9. Embedding operation kernel performance comparison of RecFlex and the baselines on two GPU platforms.

TABLE II
DETAILED V100 KERNEL ANALYSIS OF RECFLEX AND TORCHREC.

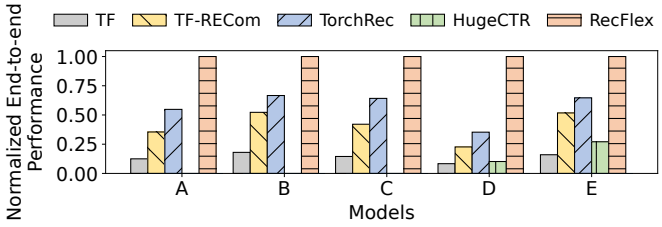
Metric Name	TorchRec	RecFlex
Memory Throughput (GB/s)	380.28	641.43
Memory Busy (%)	28.46	40.68
Max Bandwidth (%)	38.75	65.57
L1 Cache Throughput (%)	29.20	29.46
L2 Cache Throughput (%)	23.83	40.68
Avg. Active Threads Per Warp	20.35	28.54
Avg. Not Predicted Off Threads per Warp	18.13	26.03

embedding dimensions and batch sizes to saturate the GPU. Hence, HugeCTR recommends users create models with large embedding dimensions as posted in its blogs. However, large embedding dimensions for all features cannot meet the real demand of many businesses. Many features are less important than others, so using a large dimension for them leads to slower convergence and memory wastage. Besides, batch sizes during inference are often moderate due to latency constraints.

Detailed analysis. We utilize NVIDIA Nsight Compute for a detailed analysis to understand why kernels generated by RecFlex surpass others in performance. We collect some metric values of RecFlex and TorchRec by running a specific batch of model A on a V100 GPU and show them in Table II. As embedding operations are memory-intensive, we first compare the memory utilization-related metrics of RecFlex and TorchRec. We observe that RecFlex excels in all memory utilization metrics by using more suitable schedules across features. For instance, RecFlex achieves 1.69 times the memory throughput of TorchRec, indicating superior GPU DRAM utilization. Despite this, there is still a gap between RecFlex's memory bandwidth usage and the GPU's peak capacity. This is because the embedding operations are irregular and have dynamic workloads, so all the provided schedule candidates cannot fully utilize the GPU memory bandwidth. With more schedule templates, there is potential for RecFlex to find more



(a) Performance on a V100 GPU.



(b) Performance on an A100 GPU.

Fig. 10. End-to-end performance comparison of RecFlex and the baselines on two GPU platforms.

efficient GPU kernels. Additionally, we report the average number of active threads and non-predicted off threads per warp in the table. For these thread utilization-related metrics, TorchRec has much lower values than RecFlex. The reason is that the uniform schedule ignores the various per-feature workloads, causing many threads within a warp to be inactive due to early exit or be predicted off due to warp divergence.

Dataset with an extremely large number of features. To verify the scalability of RecFlex, we synthesize an extra dataset with 10,000 features. Experiments show that RecFlex still achieves a 4.2× speedup over TorchRec with this dataset.

MLPerf dataset. MLperf [38] uses TorchRec [17] as its backend and provides a synthesized multi-hot dataset based on criteo [39]. This dataset comprises only 26 feature fields and exhibits low inter-feature heterogeneity. According to our experiments, despite this dataset's low feature heterogeneity, RecFlex still achieves nearly the same kernel performance as TorchRec. In many real-world applications, feature heterogeneity is much more significant [11], [13] so that RecFlex can outperform TorchRec by selecting distinct schedules.

C. End-to-end Model Performance

We add an MLP layer with the hidden unit numbers 1024, 256, and 128 next to the embedding layer of each model. We then present the end-to-end model execution time in Figures 10(a) and (b). For HugeCTR, as we discussed in Section VI-B, it has many GPU cache-related operations that are not involved by other frameworks, so we only consider the execution time of the part next to its cache-related operations. Experimental results show that RecFlex established 7.74×, 2.69×, 6.76×, and 1.85× average speedups over TensorFlow, RECom, HugeCTR, and TorchRec on the two GPU platforms. The average end-to-end speedups of RecFlex are less than the kernel performance speedups because RecFlex does not focus on optimizing the DNN parts.

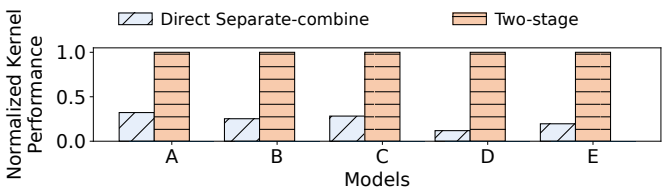


Fig. 11. Tuning result performance comparison between the two-stage tuning and the direct separate-combine approach.

D. Ablation Study

In this section, we run variants of RecFlex to evaluate its different components or design choices.

Two-stage interference-simulated schedule tuning. We replace the two-stage tuning strategy described in Section IV-A2 with the straw-man solution 1 in Section II-C, which is a direct separate-combine approach. This approach tunes the per-feature schedule directly based on the separate latency without considering inter-feature interference. It generates a non-padded kernel for each schedule candidate and measures its execution time to compare the schedule with others. We compare the tuning results of RecFlex with this variant in Figure 11. The results show that the kernels found by the two-stage tuning approach outperform the ones found by the latency-based direct approach across all models, with an average improvement of 4.82×. This significantly demonstrates the superiority of RecFlex's schedule tuning strategy.

Effectiveness of the schedule tuner. To further demonstrate the practicability and effectiveness of the schedule tuner, we randomly pick three features in model A, and for each feature, we replace its selected schedule with other schedule candidates to generate new fused embedding kernels. We then compare the performance of these kernels to see if the kernel with the schedules tuned by RecFlex can outperform others. Figure 12 presents the performance variation curve of these kernels. The points marked with "o" are the schedules tuned by RecFlex. We observe that RecFlex effectively figures out the schedules that lead to the optimal or near-optimal (the gap is extremely slight and can be caused by measurement errors) kernel performance. Besides, the figure shows that using schedules 0-20 for feature 0 and feature 2 causes significant kernel performance degradation. We find the reason is that these schedules use many registers per thread, so constraining the occupancy causes serious register spilling problems, introducing extra GPU DRAM access overheads.

Runtime thread mapping. We adjust the fusion compiler of RecFlex to replace the runtime thread mapping of the fused kernel with static ones. Specifically, we use two static thread mapping strategies, which allocate GPU blocks to each feature based on the average and maximum historical workload, respectively. We first run the runtime thread mapping kernels to collect the thread block usages and then use this historical information to determine the static thread mapping. During execution, each allocated block adjusts its corresponding computation based on the input workload. For example, if we

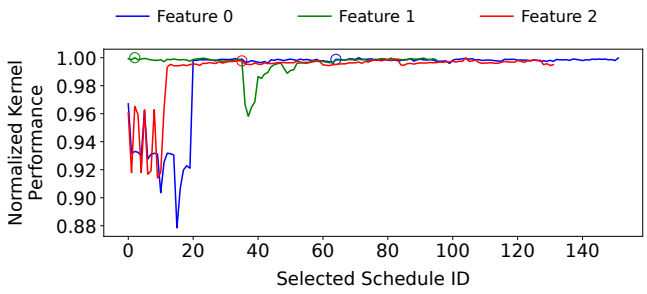


Fig. 12. Performance variation with the change of the selected schedule of a specific feature.

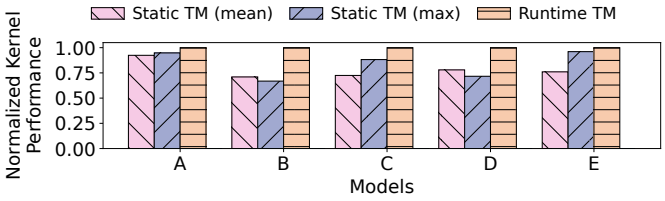


Fig. 13. Performance comparison of runtime thread mapping and two static thread mapping strategies.

allocate two blocks to a specific feature at compile time while the input workload requires three blocks at runtime, the first block will perform the computation of two blocks sequentially. Figure 13 shows that, compared with these static strategies, runtime thread mapping of RecFlex offers up to 1.41× and 1.50× improvements, because static thread mapping causes resource wastage and workload imbalance.

Furthermore, static thread mapping cannot handle the long tails of requests [5] well. For the dataset used in our evaluation, we limit the maximum batch size of a request to 512, as it is common for industrial serving systems to split batches exceeding a specific threshold. However, there are serving systems like DeepRecSys [5] that do not split batches. We generate an additional request with 2,560 samples to simulate the long tail requests in this case, and the experimental results show that for this request, the two static thread mapping strategies cause 50.5% and 40.4% performance degradation compared with runtime thread mapping.

E. Overhead Analysis

Compile and tuning overhead. RecFlex tunes the optimal schedules in $O(F \cdot K + K)$ time, which is discussed in Section IV-A. As the per-feature schedule tuning in the local stage is independent, we can simultaneously compile and tune different features. In our evaluation, we use eight individual GPUs to tune schedules and 32 CPU threads to compile generated GPU kernels in parallel. In this way, we can get the optimal fused kernel in several hours, which is acceptable as we can serve the model for a long time (e.g., one week) after the tuning. There are still potential ways to accelerate the tuning procedure, such as eliminating redundant compilation and data loading, which are not the concentration of this paper.

Runtime overhead. Host-side workload analysis can introduce some overhead. However, the involved operations are so lightweight that our experiments show that the overhead is less than 0.1% of the data loading time. This overhead can be hidden behind the entire inference pipeline.

VII. Discussions

Larger model sizes. RecFlex concentrates on the optimization of the fused embedding operations on a single GPU. To support models with embedding table sizes exceeding the GPU memory capacity, we can combine RecFlex with existing solutions. For example, we can place different embedding tables on multiple GPUs through heuristics [14] or machine learning methods [40], [41] and then use RecFlex to optimize the embedding operations on each GPU. We can also use the GPU to serve as the hot-embedding cache of the CPU [13], [14], [32], [37] by developing corresponding schedules with unified memory (UVM) [31].

Larger fusion scopes. In this paper, we explore the schedule optimization of the embedding operations [14], i.e., the embedding lookup and the following pooling operations. As we adopt a compiler approach to generating the fused kernel, there is an opportunity to cluster more operators into it. For example, some models can contain several numerical preprocess operators before the embedding operations [12]. However, involving more operators in the fused kernel challenges the schedule tuning, as we need to consider both intra- and inter-feature interference. We leave this as one of our future work.

Automatic scheduling. Currently, we require users to manually write the schedule templates to feed RecFlex. There are auto-schedulers [21], [22], [25], [42], [43] that generate and search the efficient schedule of a single operator automatically, but they do not support dynamic shape workloads yet. If they support dynamic shapes in the future, we can adjust the generated schedules of these auto-schedulers to be compatible with RecFlex's input templates, providing opportunities for further performance improvement.

VIII. Related Work

Recommendation model optimization. Extensive works [44]–[51] have been proposed to optimize the training or serving of recommendation models from different aspects. Kraken [49] is a large-scale recommendation model training system with a specialized parameter server and sparse-aware optimizers. TT-Rec [44] and EL-Rec [45] apply tensor-train compression to reduce the embedding table size. These works do not involve the GPU kernel optimization of embedding operations, which is the main focus of this paper. There are works [11], [13], [14], [32], [37], [52]–[55] rely on intra-feature heterogeneity, i.e., the heterogeneity across embedding entries of a feature, to design their systems. In contrast, RecFlex explores inter-feature heterogeneity, which is not considered by previous works. Another line of works [56]–[59] designs specialized hardware to accelerate the execution of embedding operations rather than utilize the commonly used

GPUs. TorchRec [17] and NVIDIA HugeCTR [18] are recommendation model libraries with highly manually optimized GPU kernels, but they do not consider the significant feature heterogeneity of industrial recommendation models. RecFlex further improves the GPU kernel performance by applying distinct schedules across features.

Machine learning compilers. Machine learning compilers [19], [23], [24], [26], [42], [60]–[66] are widely used to optimize the performance of model inference by performing graph transformations and generating efficient codes. These works mainly focus on optimizing classical DNNs and are complementary to RecFlex, as we can use them and RecFlex to optimize the different parts of a recommendation model. Among them, Rammer [26] fuses operators through inter- and intra-operator co-scheduling and tunes schedules in its fused kernels based on basic heuristics, but its fusion and schedule tuning cannot meet the requirements of embedding optimizations. It overlooks interference among schedules, assumes homogeneous blocks for an operator, and lacks support for dynamic workloads. RECom [12] is an end-to-end optimizing compiler for recommendation models. It fuses operations within thousands of embedding subgraphs into a single GPU kernel, but it treats different features equally in the fused kernel and uses static thread mapping, overlooking the feature heterogeneity and workload variance. Overall, previous compilers either generate inefficient kernels or fail to generate kernels for recommendation embedding operations.

IX. Conclusion

In this paper, we identify the limitation of existing recommendation model libraries and optimizing compilers, which is that they treat all features with identical schedules in their fused embedding kernels, overlooking the significant feature heterogeneity. To address this limitation, we introduce RecFlex to optimize the fused kernel with flexible schedules for different features. RecFlex tunes the generally optimal schedules based on recent input data. It first tunes the per-feature schedule in the local stage with explicitly controlled occupancy and simulated inter-feature interference, and then determines the optimal occupancy value in the global stage. RecFlex also incorporates a compiler to generate fused GPU kernels with heterogeneous schedules. The thread mapping of the fused kernels is determined at runtime to reduce workload imbalance and resource wastage. Our evaluation shows that RecFlex outperforms existing solutions significantly. As far as we know, RecFlex is the first work to explore schedule tuning in the fused kernel with inter-schedule interference. We hope our work can inspire future research on this new topic.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 62322213 and 62172419) and Beijing Nova Program (No. 20230484397 and 20220484137). We sincerely thank all the anonymous reviewers for their insightful comments and feedback. We would also thank Zheng Wang at UCSD and Jiawei Guan at RUC for their extensive suggestions. We use ChatGPT to polish the writing. Zaifeng Pan, Feng Zhang, Ruofan Wu, 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.

References

[1] H. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, A. Karatzoglou, B. Hidasi, D. Tikk, O. S. Shalom, H. Roitman, B. Shapira, and L. Rokach, Eds. ACM, 2016, pp. 7–10.

[2] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, S. Sen, W. Geyer, J. Freyne, and P. Castells, Eds. ACM, 2016, pp. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

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

[4] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 488–501.

[5] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 982–995.

[6] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1059–1068.

[7] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. 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*. AAAI Press, 2019, pp. 5941–5948.

[8] M. Cheng, Y. Gao, G. Liu, H. Jin, and X. Zhang, "Easyrec: An easy-to-use, extendable and efficient framework for building industrial recommendation systems," *CoRR*, vol. abs/2209.12766, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2209.12766

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

[10] Q. Wang, Z. Ji, H. Liu, and B. Zhao, "Deep bayesian multi-target learning for recommender systems," *CoRR*, vol. abs/1902.09154, 2019. [Online]. Available: http://arxiv.org/abs/1902.09154

[11] F. Lai, W. Zhang, R. Liu, W. Tsai, X. Wei, Y. Hu, S. Devkota, J. Huang, J. Park, X. Liu *et al.*, "{AdaEmbed}: Adaptive embedding for {Large-Scale} recommendation models," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 817–831.

[12] Z. Pan, Z. Zheng, F. Zhang, R. Wu, H. Liang, D. Wang, X. Qiu, J. Bai, W. Lin, and X. Du, "RECom: A Compiler Approach to Accelerating Recommendation Model Inference with Massive Embedding Columns," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 268–286.

[13] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C. Wu, "Recshard: statistical feature-based memory optimization for industry-scale neural recommendation," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 344–358. [Online]. Available: https://doi.org/10.1145/3503222.3507777

[14] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. R. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 993–1011. [Online]. Available: https://doi.org/10.1145/3470496.3533727

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX OSDI 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 265–283.

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

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

[19] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 578–594.

[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[21] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 3393–3404. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

[22] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. 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*. USENIX Association, 2020, pp. 863–879. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zheng

[23] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai, S. L. Song, and W. Lin, "Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 359–373.

[24] Z. Zheng, Z. Pan, D. Wang, K. Zhu, W. Zhao, T. Guo, X. Qiu, M. Sun, J. Bai, F. Zhang, X. Du, J. Zhai, and W. Lin, "BladeDISC: Optimizing Dynamic Shape Machine Learning Workloads via Compiler Approach," in *Proc. ACM Manag. Data 1, 3 (SIGMOD), Article 206 (September 2023), 29 pages*. ACM, 2023.

[25] Y. Ding, C. H. Yu, B. Zheng, Y. Liu, Y. Wang, and G. Pekhimenko, "Hidet: Task-mapping programming paradigm for deep learning tensor programs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 370–384.

[26] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. 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*. USENIX Association, 2020, pp. 881–897. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ma

[27] X. Zhao, M. Jahre, and L. Eeckhout, "Hsm: A hybrid slowdown model for multitasking gpus," in *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*, 2020, pp. 1371–1385.

[28] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 653–663.

[29] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: a versatile programming framework for pipelined computing on GPU," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, Eds. ACM, 2017, pp. 587–599.

[30] Zhen Zheng and Chanyoung Oh and Jidong Zhai and Xipeng Shen and Youngmin Yi and Wenguang Chen, "Hiwaylib: A software framework for enabling high performance communications for heterogeneous pipeline computations," in *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*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 153–166.

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

[32] Y. Zhang, L. Chen, S. Yang, M. Yuan, H. Yi, J. Zhang, J. Wang, J. Dong, Y. Xu, Y. Song, Y. Li, D. Zhang, W. Lin, L. Qu, and B. 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*. IEEE, 2022, pp. 3453–3466. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00324

[33] NVIDIA, "Thrust: The C++ Parallel Algorithms Library," https://github.com/NVIDIA/thrust, 2023.

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

[35] Kaggle, "Click-Through Rate Prediction," https://www.kaggle.com/c/avazu-ctr-prediction, 2023.

[36] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy, "Fbgemm: Enabling high-performance low-precision deep learning inference," *arXiv preprint arXiv:2101.05615*, 2021.

[37] M. Xie, Y. Lu, J. Lin, Q. Wang, J. Gao, K. Ren, and J. Shu, "Fleche: an efficient gpu embedding cache for personalized recommendations," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 402–416.

[38] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain,*

May 30 - June 3, 2020. IEEE, 2020, pp. 446–459. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00045

[39] C. A. Lab, "Download Criteo 1TB Click Logs dataset," https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset, 2023.

[40] D. Zha, L. Feng, Q. Tan, Z. Liu, K.-H. Lai, B. Bhushanam, Y. Tian, A. Kejariwal, and X. Hu, "Dreamshard: Generalizable embedding table placement for recommender systems," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 190–15 203, 2022.

[41] D. Zha, L. Feng, B. Bhushanam, D. Choudhary, J. Nie, Y. Tian, J. Chae, Y. Ma, A. Kejariwal, and X. Hu, "Autoshard: Automated embedding table sharding for recommender systems," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 4461–4471.

[42] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, F. Yang, M. Yang, L. Zhou, A. Cidon, and G. Pekhimenko, "ROLLER: fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 233–248. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zhu

[43] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 859–873. [Online]. Available: https://doi.org/10.1145/3373376.3378508

[44] C. Yin, B. Acun, C. Wu, and X. Liu, "Tt-rec: Tensor train compression for deep learning recommendation models," in *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, A. Smola, A. Dimakis, and I. Stoica, Eds. mlsys.org, 2021.

[45] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, "El-rec: efficient large-scale recommendation model training via tensor-train embedding table," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2022, pp. 1007–1020.

[46] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo, "FAERY: an fpga-accelerated embedding-based retrieval system," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 841–856. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zeng

[47] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "AIBox: CTR prediction model training on a single node," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.

[48] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke, "Optimizing deep learning recommender systems training on cpu cluster architectures," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[49] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. Ao, W. Xu, and J. Shu, "Kraken: memory-efficient continual learning for large-scale real-time recommendations," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–17.

[50] Z. Wang, Y. Wang, J. Deng, D. Zheng, A. Li, and Y. Ding, "Rap: Resource-aware automated gpu sharing for multi-gpu recommendation model training and input preprocessing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 964–979.

[51] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: a high-performance framework for enabling near orthogonal processing on compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2022.

[52] X. Miao, Y. Shi, H. Zhang, X. Zhang, X. Nie, Z. Yang, and B. Cui, "HET-GMP: A graph-based system approach to scaling large embedding model training," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 470–480.

[53] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Accelerating recommendation system training by leveraging popular choices," *Proc. VLDB Endow.*, vol. 15, no. 1, pp. 127–140, 2021.

[54] S. Agarwal, C. Yan, Z. Zhang, and S. Venkataraman, "Bagpipe: Accelerating deep recommendation model training," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 348–363.

[55] Z. Wang, Y. Wang, B. Feng, G. Huang, D. Mudigere, B. Muthiah, A. Li, and Y. Ding, "OPER: Optimality-Guided embedding table parallelization for large-scale recommendation model," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 667–682. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/wang

[56] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. M. Hazelwood, B. Jia, H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. Wu, M. Hempstead, and X. Zhang, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 790–803.

[57] W. Jiang, Z. He, S. Zhang, T. B. Preußer, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang, and G. Alonso, "Microrec: Efficient recommendation inference by hardware and data structure solutions," in *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, A. Smola, A. Dimakis, and I. Stoica, Eds. mlsys.org, 2021.

[58] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "Rm-ssd: In-storage computing for large-scale recommendation inference," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1056–1070.

[59] H. Liu, L. Zheng, Y. Huang, C. Liu, X. Ye, J. Yuan, X. Liao, H. Jin, and J. Xue, "Accelerating personalized recommendation with cross-level near-memory processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

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

[61] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[62] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[63] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for GPU kernels," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, J. W. Lee, S. Hack, and T. Shpeisman, Eds. IEEE, 2022, pp. 14–27. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741270

[64] G. Huang, Y. Bai, L. Liu, Y. Wang, B. Yu, Y. Ding, and Y. Xie, "Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[65] Z. Chen, A. Kerr, R. Cai, J. Kosaian, H. Wu, Y. Ding, and Y. Xie, "Evt: Accelerating deep learning training with epilogue visitor tree," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 301–316.

[66] D. Zhuang, Z. Zheng, H. Xia, X. Qiu, J. Bai, W. Lin, and S. L. Song, "MonoNN: Enabling a new monolithic optimization space for neural network inference tasks on modern GPU-Centric architectures," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 989–1005. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhuang

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 This paper proposes RecFlex, a recommendation model optimization system that optimizes the embedding operations by selecting distinct schedules for different feature fields within the fused GPU kernel. RecFlex outperforms state-of-the-art recommendation model libraries and compilers significantly.
- C_2 The *interference-aware schedule tuner* within RecFlex proposes a two-stage interference-simulated schedule tuning, which tunes per-feature schedules effectively and outperforms the direct separate-combine approach.
- C_3 The *heterogeneous schedule fusion compiler* within RecFlex proposes runtime thread mapping for generated GPU kernels, which can handle dynamic workloads efficiently and outperform the two static thread mapping strategies.

B. Computational Artifacts

A_1 https://zenodo.org/doi/10.5281/zenodo.12158626

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 9-10 Table 2
A_1	C_2	Figures 11
A_1	C_3	Figure 13

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact contains the source codes of the entire RecFlex system, including the components *interference-aware schedule tuner* and *heterogeneous schedule fusion compiler*.

Expected Results

- C_1 RecFlex should be faster than TensorFlow, RECom, HugeCTR, and TorchRec on the GPU.
- C_2 The GPU kernels generated by the two-stage interference-simulated schedule tuning should be faster than the ones tuned by the direct separate-combine approach.
- C_3 The GPU kernels with runtime thread mapping should be faster than the ones with static thread mapping strategies.

Expected Reproduction Time (in Minutes)

The expected dataset generation time is 20 min. The expected schedule tuning time of RecFlex for each dataset on DGX-1 is 240 min. The expected execution time of RecFlex is 5 min. The expected setup and execution time of the baselines is about 180 min.

Artifact Setup (incl. Inputs)

Hardware: We use NVIDIA V100 and A100 GPUs in the evaluation. Using multiple GPUs (e.g., DGX-1 with eight GPUs) can accelerate the schedule tuning time. The minimum required disk space is 600GB.

Software: The Python and CUDA versions used in the evaluation are 3.8 and 11.8, respectively. For the evaluation of RecFlex and TorchRec, we use PyTorch with the version 2.2.0. The TorchRec version is 0.6.0 with FBGEMM 0.6.0. For TensorFlow and RECom, we use TensorFlow 2.6.2, which is the version supported by RECom. For HugeCTR, we use its official image with version 23.04.

The corresponding URLs are:

- PyTorch: https://github.com/pytorch/pytorch
- TorchRec: https://github.com/pytorch/torchrec
- TensorFlow: https://github.com/tensorflow/tensorflow
- RECom: https://github.com/AlibabaResearch/recom
- HugeCTR: https://github.com/NVIDIA-Merlin/HugeCTR

Datasets / Inputs: Run 'SC_artifact/datagen.sh' to call the script 'data_synthesis/data_generate.py' to generate all the datasets used in our evaluation. This script generates the dataset based on given per-feature data distribution and table shape configurations, which can be found in each sub-folder under 'examples/models'.

Installation and Deployment: We use the docker image 'nvidia/cuda:11.8.0-cudnn8-devel-ubuntu20.04' as the basic execution environment. After launching the corresponding container, please follow the instructions in 'README.md' to set up the environment, including installing Python 3.8, PyTorch 2.2.0, CMake 3.28.3, and NVIDIA Nsight System 2023.4.1. Alternatively, users can also use the Dockerfiles under 'SC_artifact' to construct the images, which already contain the dependent packages.

Artifact Execution

Individual tasks:

- T_1 Generate datasets A-E.
- T_2 Use RecFlex to tune schedules and then benchmark the final generated kernels.
- T_3 Benchmark TensorFlow and RECom.
- T_4 Benchmark TorchRec.
- T_5 Benchmark HugeCTR with models D and F.
- T_6 Tune schedules by the direct separate- combine approach and then benchmark newly generated kernels.

T_7 Generate kernels with static thread mapping and then benchmark them.

T_8 Plot all figures.

Note that all the experimental parameters are provided in the execution scripts.

Dependencies:

D_1 $T_1 \rightarrow T_2, T_3, T_4, T_5, T_6$

D_2 $T_2 \rightarrow T_7$

D_3 $T_2, T_3, T_4, T_5, T_6, T_7 \rightarrow T_8$.

Artifact Analysis (incl. Outputs)

Figures 9-13 in the paper will be generated.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

We have provided two Dockerfiles to allow users to prepare the execution environments, where one is for RecFlex and TorchRec, and the other is for TensorFlow and RECom.

The instructions to build Docker images are:

```
cd SC_artifact
docker build -f ./Dockerfile -t recflex:latest .
docker build -f ./TF.Dockerfile -t recom:latest .
```

Then, after launching the containers, run the following commands in the code directory to install RecFlex and generate datasets:

```
pip install .
./post_install.sh # only required in recflex container
./SC_artifact/datagen.sh
```

We have integrated these setup and execution commands within a script, so you do not need to run each command manually.

Artifact Execution

We have integrated the setup and execution commands within 'SC_artifact/run_all.sh', which can reproduce figures 9-10 in the paper directly.

```
./SC_artifact/run_all.sh
```

Note that the script assumes there are four GPUs available. If the number of GPUs is not four, the environment variable `CUDA_VISIBLE_DEVICES` used in 'SC_artifact/run_recflex.sh' needs to be adjusted accordingly.

In detail, this script contains the following execution steps:

1) Setup the execution environment and prepare the dataset as illustrated in *Artifact Setup*.
2) Run 'SC_artifact/run_recflex.sh' to use RecFlex to tune the optimal schedules based on tuning data and then use Nsight System to profile the execution time of the generated kernels on the test data.
3) Run 'SC_artifact/run_torchrec.sh' to profile the execution time of TorchRec.
4) Run 'SC_artifact/run_tf_recom.sh' to profile the execution time of TensorFlow with and without the compilation optimization of RECom.
5) Run 'SC_artifact/run_hugectr.sh' in HugeCTR's official container to profile the execution time of HugeCTR.
6) Run 'SC_artifact/plot.sh' to generate the figures based on the data collected in previous steps.

Artifact Analysis (incl. Outputs)

After executing the script, there will be two figures, 'kern.pdf' and 'e2e.pdf', under the directory 'SC_artifact/outputs'. They correlate figures 9 and 10 in the paper, respectively.

These figures illustrate the normalized embedding kernel and model execution performance of RecFlex and the baselines. The expected results are that RecFlex can achieve the highest performance among these systems.