Efficient Inter-Operator Scheduling for Concurrent Recommendation Model Inference on GPU

Shuxi Guo[1,2], **Zikang Xu**[1], **Jiahao Liu**[1], **Jinyi Zhang**[1], **Qi Qi**[1], **Haifeng Sun**[1], **Jun Huang**[2], **Jianxin Liao**[1,*] and **Jingyu Wang**[1,3,*]
[1]State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications, Beijing, China
[2]Meituan, Beijing, China
[3]Pengcheng Laboratory, Shenzhen, China
{sx, xuzikang, liujiahao, elementp,qiqi8266,hfsun}@bupt.edu.cn, huangjun03@meituan.com,
{liaojx,wangjingyu}@bupt.edu.cn

Abstract

Deep learning-based recommendation systems are increasingly important in the industry. To meet strict SLA requirements, serving frameworks must efficiently handle concurrent queries. However, current serving systems fail to serve concurrent queries due to the following problems: (1) inefficient operator (op) scheduling due to the query-wise op launching mechanism, and (2) heavy contention caused by the mutable nature of recommendation model inference. This paper presents RecOS, a system designed to optimize concurrent recommendation model inference on GPUs. RecOS efficiently schedules ops from different queries by monitoring GPU workloads and assigning ops to the most suitable streams. This approach reduces contention and enhances inference efficiency by leveraging inter-op parallelism and op characteristics. To maintain correctness across multiple CUDA streams, RecOS introduces a unified asynchronous tensor management mechanism. Evaluations demonstrate that RecOS improves online service performance, reducing latency by up to 68%.

1 Introduction

Deep neural networks (DNNs) based recommendation models (RMs) have been widely deployed in many large companies, including Google [Zhao *et al.*, 2019; Covington *et al.*, 2016],Alibaba [Zhou *et al.*, 2018a; Zhou *et al.*, 2019], and Netflix [Koren *et al.*, 2009]. An RM typically consists of a memory-intensive part, known as the embedding lookup layer and often contains caches, and a compute-intensive part, known as several fully connected layers. Efficient inference of recommendation models is crucial for the profits of an online service.

While an RM does not require massive computing resources, it still faces extremely high concurrency and strict latency requirements in online serving. Recently, online service frameworks such as TensorFlow Serving (TS) [Olston *et al.*, 2017b], and Triton Inference Server [NVIDIA, 2024c] have emerged. When an op finishes execution, the online service system determines the execution order of ops based on the topology of the computation graph. Then, ops are launched to GPU in a First Come First Served (FCFS) manner. Such a workflow, however, faces poor performance in concurrent queries scenarios. As illustrated in Figure 1, the inference latency of processing thirty concurrent queries increases by 1 to 6 times compared to processing only one query. This occurs because multiple queries independently launch ops to the GPU during traffic peaks, leading to unavoidable op interleaving, forcing ops from different queries to alternate on the GPU. Furthermore, the service frameworks typically launches all ops to the same CUDA Stream (stream) [NVIDIA, 2024a]. This mechanism ensures that ops are executed in the correct order by utilizing the stream's features. However, as indicated in Figure 3, this approach does not fully utilize GPU resources and leads to prolonged latency.

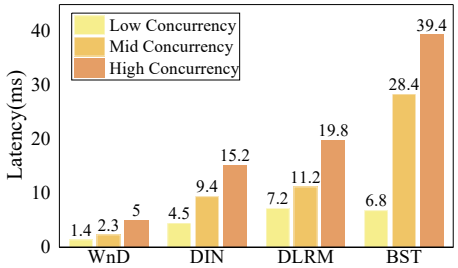

Figure 1: Online inference performance was evaluated at three levels of concurrency: low (1 client), medium (15 clients), and high (30 clients). Results showed that under high concurrency, the inference latency can increase by as much as 4.7 times compared to the low concurrency scenario (BST).

It's straightforward to use multiple streams to serve concurrent queries. However, as shown in Figure 3, such a mechanism is still suboptimal. Unlike other DNN models' infer-

[0]*Corresponding authors

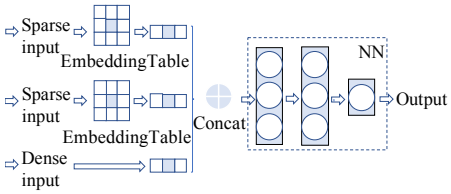

Figure 2: RMs consist of an embedding part (embedding tables) and a dense part (neural networks).

ence, RM's inference is mutable. When accessing the mutable part of RM, locks are needed to guarantee the correctness. Thus, when multiple ops access the same mutable region, conflicts occur and ops' execution time increases. What's worse, synchronization between ops from multiple streams is needed to ensure the correctness of the inference, which further increases the overhead.

To address the problems listed above and improve the online execution efficiency of RMs, we introduce RecOS, an online RM scheduling system. We adopted Nvidia's Multistream [NVIDIA, 2024a] mechanism and scheduled ops across multiple streams to fully utilize GPU resources while minimizing the overhead caused by disordered op scheduling and conflicts. To achieve highly efficient op scheduling on multiple streams, RecOS must obtain real-time information about the workload of each stream. Since Nvidia does not provide a related API, RecOS establishes a GPU workload monitor. This monitor estimates the queuing status of each stream by considering ops' execution order and computational attributes such as grid size and execution time. After obtaining the queuing status of each stream, RecOS uses greedy algorithms to assign ops to suitable streams based on op's type. Finally, RecOS introduces a unified asynchronous tensor management mechanism based on characteristic of tensors' lifetime to eliminate the synchronization overhead.

Our experiments on multiple RMs show that RecOS can effectively improve the inference latency. Our algorithm can significantly reduce inference latency under high concurrency. Compared to current service frameworks, RecOS can achieve an inference latency improvement of up to 68%.

Our contributions are summarized as follows:

1. We analyzed RMs' computation graph characteristics, mutable nature, and execution process in concurrent scenarios, identifying key opportunities for optimization through scheduling. To the best of our knowledge, we are the first to highlight the mutable nature of RMs and its impact on concurrent inference.

2. We introduce RecOS, an op-level inference scheduling system specifically tailored for RMs, designed to improve kernel scheduling efficiency and reduce inference latency.

3. We extensively tested multiple RMs under various online concurrent scenarios, demonstrating the effectiveness of our approach.

Model	#Ops	Grid Ratio $[0, 28]$	Grid Ratio $(28, \infty]$	Parallel Op Ratio
RMs	350	85%	15%	78%

Table 1: Model Characteristics

2 Background and Motivation

2.1 Architecture and Characteristics of RMs

Figure 2 sketches the architecture of RMs. RMs primarily accept two types of input: sparse features (such as *user ID* and *video ID*) and dense features (such as *user age* and *video click count*). The sparse module of RMs lookups embedding table, whose size may reach hundreds of GBs, to transforms sparse features into dense features. The feature interaction module is responsible for combining the dense and sparse features. Finally, the prediction module takes the output from the interaction module and calculates the click-through rate (CTR) probability for each user-item pair using neural network computations. The CTR probabilities for all items are then ranked, and the top-N choices are presented to the user.

Owing to their unique hybrid structure, RMs' inference possess the following characteristics:

1. **Low Computational Complexity**

   Unlike many well-known neural networks, RMs generally have lower computational complexity. Based on our statistical analysis across multiple RMs[Cheng *et al.*, 2016; Zhou *et al.*, 2018b; Chen *et al.*, 2019; Naumov *et al.*, 2019], Table 1 summarizes their common characteristics, the 'Grid Ratio [0,28] = 85%' indicates that 85% of the ops have a grid size of 28 or less, so each op by itself cannot fully occupy all SMs on an A30 GPU.[1].

2. **High Parallelism** We abstract the computational graph of an RM model as a directed acyclic graph (DAG). For any node $\mathbf{a}$ within this graph, if there exists another node $\mathbf{b}$ that does not communicate with $\mathbf{a}$, then $\mathbf{a}$ and $\mathbf{b}$ can be executed in parallel. As shown in Table 1, up to 78% of RMs' ops can run in parallel with other ops.

3. **Mutable Inference** Unlike conventional models, RM weights are not fully frozen during inference. Because of their large embedding tables, RMs often employ GPU caches [Wang *et al.*, 2022; Xie *et al.*, 2022] to improve memory efficiency. The cache records metadata (e.g., access frequency) of embeddings and updates HBM parameters from external memory (e.g., DRAM) according to caching strategies. To prevent accessing invalid cache entries, caches use locks to protect data. Because these caches update metadata on each access, multiple embedding ops may compete for the same entry, causing lock contention and prolonged execution time if scheduled improperly.

[1]Nvidia A30, a widely used AI inference card, contains 56 SMs

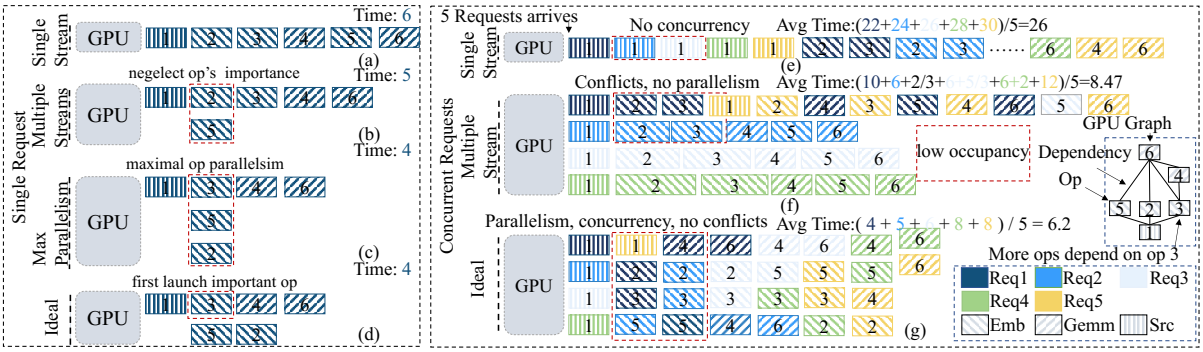

Figure 3: An example of GPU op scheduling with different numbers of queries, streams, and launch strategies. The ops with north-west/south-east hatching, north-east/south-west hatching, and vertical hatching represent embedding ops with cache, computation ops, and source ops receiving inputs, respectively. (a)- (d) handle one query, while (e)- (g) handle five concurrent queries. The GPU utilizes one stream in a) and (e), two streams in (b) and (d), three streams in (c), and four streams in (f) and (g). It is assumed that the ops in the GPU graph are identical and consume less than 1/4 of the GPU's SMs. In figure (f), multiple queries access the same embedding cache, leading to prolonged latency.

2.2 Analysis of Current Schedule

Single Query

We first consider scheduling within a single query.

Single Stream. The op scheduler launches all ops onto one stream based on topological sorting, as shown in Figure 3 (a). This method results in the longest inference latency and the lowest resource utilization, since inter-op parallelism is not exploited.

Multiple Streams. To utilize inter-op parallelism, it is necessary to launch ops onto multiple streams. Figures 3 (b) and 3 (c) illustrate the use of 2 and 3 streams, achieving an inference latency improvement of 1/6 and 1/3, respectively.

Ideal. Although increasing the number of streams improves inference latency, it does not always achieve optimal resource utilization. Figure 3 (d), which utilizes 2 streams, reaches the same inference improvement as that achieved with 3 streams in Figure 3 (c). This is due to the topological sorting scheduling algorithm. If op 3 can be finished earlier, ops 4 and 6 can be launched earlier, which can reduce the overall query inference latency.

2.3 Concurrent Queries

Next, we consider the case of processing multiple concurrent queries.

Single Stream. Many existing DNN service systems, such as TS [Olston *et al.*, 2017a], utilize a single computation stream for all concurrent queries. However, as depicted in Figure 3 (e), the inference latency of the GPU graph might be substantially increased due to op interleaving (see the red dashed box), as only one kernel runs on the GPU at any given time. This method results in poor throughput of 0.167 (processing 5 queries during 30 time units).

Multiple Streams. Increasing the number of streams can mitigate the impact of op interleaving under concurrent queries. Figure 3 (f) illustrates the use of 4 streams, selecting one for each query in a round-robin manner and launching all ops of a query onto the same stream. This approach reduces the inference latency from 26 to 8.47 and increases the throughput from 0.167 to 0.417. However, it does not exploit inter-op parallelism, leading to a non-optimal inference latency and low resource utilization. Besides, the conflicts between caches increase the ops' execution time.

Ideal. Figure 3 (g) presents the ideal schedule of RMs across multiple streams. Apart from utilizing multiple streams, this also leverages inter-op parallelism (see the red dashed-line box) and avoids cache conflicts. Figure 3 (g) reaches an average inference latency of 6.2 and a throughput of 0.635, which achieves best compared with Single Stream and Multiple Streams.

2.4 Insight

From the above illustration, we can derive two insights.

Underutilized Inter-Op Parallelism. Table 1 reveals that most computation graphs have a high degree of parallelism. However, due to the one-by-one scheduling strategy shown in Figure 3 (a), (e), and (f), RM models' parallelism is underutilized.

Disordered Scheduling under Concurrency. Figure 3 (e) shows the disordered kernel execution timeline under concurrent queries. When the dependencies of a kernel from one query are satisfied, it may still have to wait because kernels from other queries have already been launched into the queue. Besides, the conflicts between caches increase the ops' execution time and increase the inference latency of total queries. The primary cause is the absence of a unified scheduling mechanism.

3 Methodology

3.1 Overview

The goal of RecOS is to provide efficient scheduling of ops across multiple streams (see Figure 3 (f)) to improve inference latency under traffic peaks. Figure 4 shows the overall framework of our system, which includes (a) an offline component, which profiles models before online service, and (b)

an online component, which schedules ops under concurrent queries.

Offline. Before online scheduling, RecOS utilizes Nsight Systems [NVIDIA, 2024b] to obtain information about the ops on the GPU in the model, including op execution time and kernel grid size.

Online. The online component comprises a Model Weights Database, a Workload Monitor, a Multistream Scheduler, and a Memory Manager.

3.2 Model Weights Database

Op Importance

To facilitate explanation, we abstract the computational graph into a DAG. Let $\mathcal{G} = (\mathcal{V}, \mathcal{L})$ denote the DAG of a DNN, where $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices representing the ops of the DNN and $\mathcal{L}$ is the set of links. A link $(v_i, v_j) \in \mathcal{L}$ indicates that v_i must be processed before v_j, with v_i feeding its output to v_j.

As discussed before in Section 2.2, scheduling ops based on their importance is more effective than using the topological order. We calculate the $DepValue$ for each op as the op's importance in the computation graph:

$$DepValue(v) = 1.0 + \sum_{(v,v_n)\in\mathcal{L}} \frac{DepValue(v_n)}{inDegree(v_n)}$$

where $inDegree(v_n)$ represents the in-degree value of the vertex v_n.

Op Execution Statistics

Op execution statistics include grid size and execution time, which can be collected using Nsight Systems [NVIDIA, 2024b].

Grid Size. The grid size refers to the number of blocks within an op and can be used to infer the maximum number of SMs that an op can occupy on the GPU. The grid size of an op is calculated as follows:

$$GridSize(O) = \sum_{k\in O} GridSize(k)$$

where $GridSize(O)$ and $GridSize(k)$ represent the grid size of an op and a kernel, respectively, and $k \in O$ indicates that the kernel belongs to an op.

Execution Time. Execution Time refers to the duration of an op's execution, including the time for kernel preparation, kernel launch, and kernel execution.

3.3 Workload Monitor

To effectively schedule ops, we need to acquire the status of ops running on the GPU. As NVIDIA does not provide APIs that meet our requirements, we implemented our own monitor to estimate the running status of the GPU (as shown in Figure 4).

Assuming op o will be launched on stream s, before its execution starts, the monitor appends it to the op information queue of stream s and removes it upon completion of execution. We use grid size and execution time as metrics for measuring the resource occupancy and execution of a stream.

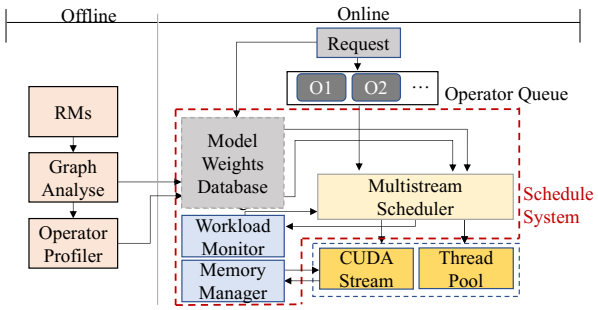

Figure 4: Architecture of RecOS.

The expected completion time for stream s is calculated as follows:

$$CompletionTime(s) = \sum_{o\in opinstream(s)} ExecTime(o)$$

where $opinstream(s)$ refers to the ops that are stored in the op info queue of stream s.

The resource occupancy at time t is calculated as follows:

$$ResOccupancy(t) = \sum_{o\in opattime(t)} GridSize(o)$$

where $opattime(s)$ refers to ops that are being executed at time t.

3.4 Multistream Scheduler

Execution Order

Computation Op. The computation op execution order is determined by both the topology and the $DepValue$. First, before computation, RecOS calculates the $DepValue$ for all ops. Once an op finishes computation, all dependent ops go into the queue. The scheduler then selects the op with the highest $DepValue$ from queue to launch; i.e., v_3 would be selected first after v_1 finishes execution, ensuring that the most critical ops are scheduled first.

Embedding Op. Because embedding ops require locks for cache consistency, concurrent embedding lookups on the same cache entry cause lock contention. RecOS utilizes a round-robin strategy to reduce contention caused by concurrent embedding operations. For each new query, RecOS determines a unique starting position. This position depends on the previous query's execution state, ensuring a staggered offset:

$$StartIdx = (NowIdx_{prev} + offset) \bmod |emb\ ops|$$

$$offset = \frac{NowIdx_{prev} - StartIdx_{prev}}{2}$$

where $NowIdx_{prev}$ is the index of the last launched embedding op, $StartIdx_{prev}$ is the initial embedding op launching position of the previous query, and $|emb\ ops|$ represents the total number of embedding ops. This approach yields two primary benefits. First, by introducing an offset, it effectively staggers similar embedding operators across concurrent queries, thereby reducing lock contention. Second, it dynamically adapts the scheduling offset according to the real-time execution status.

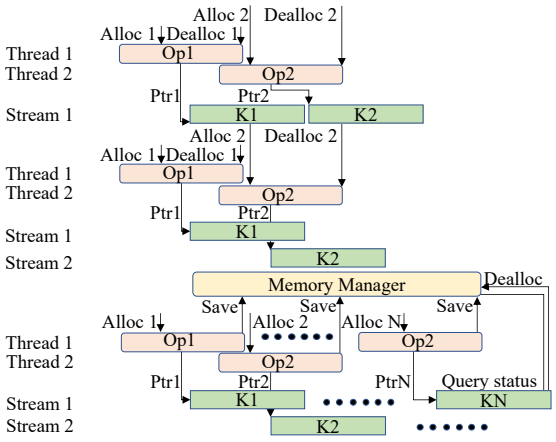

Figure 5: (a) When there is only one stream, concurrent kernel execution does not occur. (b) GPU memory might be deallocated before the kernel finishes its execution. (c) The Memory Manager deallocates memory after kernel execution.

Stream Selection

The scheduler employs different approaches for computaion ops and embedding ops.

Computation op. The scheduler uses Workload Monitor's information and op's statistics such as grid size to select the most appropriate stream. First, if any stream is idle, the scheduler immediately dispatches the computation op to that stream. Otherwise, it identifies the two streams with the shortest and second-shortest completion times, then estimates the overall workload if the op were placed on either one. If the shortest stream's estimated workload violates certain thresholds, the scheduler switches to the second-shortest stream; otherwise, it remains on the shortest.

Embedding op. For embedding ops, the scheduler focuses on avoiding cache conflicts. It first checks if there are any streams on which the op can be launched without conflict. If such streams exist, the scheduler assigns the op to the stream with the shortest expected completion time. Otherwise, it reuses the op's previously assigned stream.

3.5 Memory Manager

Next, we discuss how to ensure ops's correct execution across multiple streams through asynchronous memory management.

Ops utilize tensors to store and transfer data. To maintain the correctness of computations, it is critical to ensure that one op's memory space is not compromised by other ops.

When there is a single stream, the stream characteristic ensures that only one kernel is executed on the GPU, thereby preventing data corruption. However, as Figure 5 illustrates, when multiple streams are used, there is a risk of memory corruption, which can lead to incorrect computation results. This problem could be solved if each GPU op waits for the completion of its prior op's kernel execution after being launched. However, this synchronous approach can reduce the efficiency of kernel launches and introduce additional overhead into GPU graph computations.

To minimize the overhead associated with multistream memory management, we propose an asynchronous memory management approach for multistream RM execution. RecOS introduces a centralized memory manager to properly free GPU buffers used by ops. It utilizes a separate tensor pool for each stream to collect tensors. When a tensor pool reaches a certain threshold, the manager checks the status of corresponding kernels and releases tensors if the kernels have finished execution.

4 Experiment

4.1 Experiment Setting

Implementation. We implemented RecOS on TS [Olston *et al.*, 2017a], an open-source machine learning service system designed for production environments. It's noteworthy that RecOS can be easily integrated with other serving systems, such as Triton Inference Server [NVIDIA, 2024c].

Hardware and Software. We deployed our system on a server equipped with an Intel (R) Xeon (R) Platinum 8352Y CPU and an Nvidia A30 GPU (24 GB HBM2 and 56 SMs available), matching our production environment. All code was compiled using GCC and nvcc with the -O3 option. We used CUDA driver version 525 and CUDA Toolkit 12.0.

Models. We evaluated four representative recommendation models from in-house production: WnD [Cheng *et al.*, 2016], DIN [Zhou *et al.*, 2018b], DLRM [Naumov *et al.*, 2019], and BST [Chen *et al.*, 2019]. These models represent different architectural paradigms in modern recommendation systems and are widely deployed in production environments. It is noteworthy that RMs comprise many small kernels, which leads to high kernel launch overhead. Therefore, in a production setting, RMs would first be optimized by tools such as TVM [Chen *et al.*, 2018] to reduce such overhead. To closely mimic the production environment, we applied such optimizations.

Metrics. We selected latency as the main metric. Latency refers to the duration of model inference, excluding the time consumed for serialization, deserialization, and the network delays associated with sending and receiving queries. Experiments were conducted at five levels of concurrency: 1, 8, 15, 22, and 30. The clients keep sending queries after receiving the response of the previous queries from the server[2]. Besides, we also simulated online traffic to test the performance.

Comparison. We compared the performance of our system with the following three approaches: (1) TSSS, the default Single Stream kernel launch mechanism of TS [Olston *et al.*, 2017a]; (2) TSMS, the MultiStream implementation by TS [Olston *et al.*, 2017a], which contains multiple computation streams and launches kernels from one query to multiple streams based on the topology of the computation graph; (3) MSSS, a scheme that launches all kernels belonging to one query onto the same stream, as illustrated in Figure 3 (f); (4)

[2]We omit throughput measurement because under our client-query pattern (where each client sends the next query only after receiving the previous response), the system throughput is essentially the reciprocal of latency.

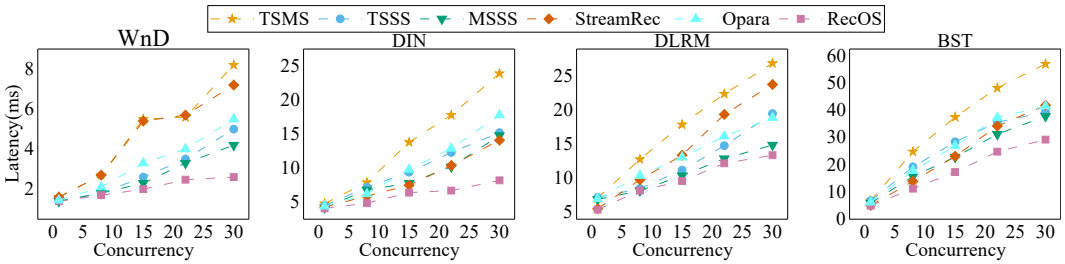

Figure 6: Runtime Performance of RecOS on Four Models. At low concurrency levels, RecOS offers only moderate improvements compared to TSSS, TSMS, and MSSS. However, when concurrency reaches a high level, RecOS's latency is lower than that of all other schemes.

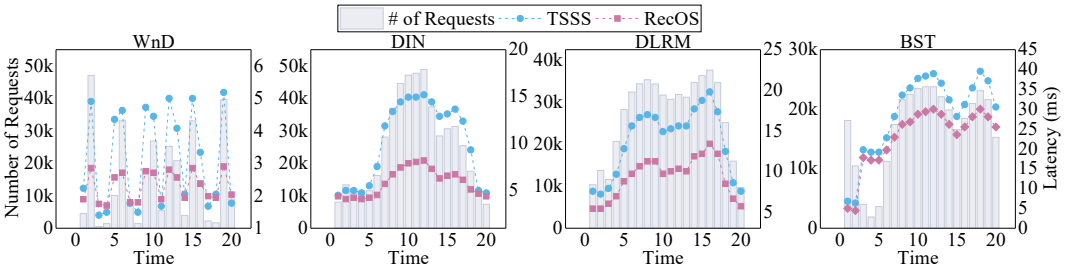

Figure 7: Runtime Performance under Online Traffic: (a) Pulse-Type Traffic; (b) Unimodal-Type Traffic; (c) (d) Bimodal Traffic. Among all traffic types, RecOS reduces inference latency during traffic peaks.

StreamRec, the Stream Assignment method for Recommendation models proposed in [Niu *et al.*, 2023]; (5) Opara, a resource- and interference-aware DNN op parallel scheduling framework to accelerate DNN inference on GPUs [Chen *et al.*, 2024].

4.2 Overall Performance

This section evaluates RecOS using the four optimized models, comparing it with the five aforementioned scheduling baselines: TSSS, TSMS, MSSS, StreamRec, and Opara. Figure 6 presents the comparison of inference latency.

Figure 6 shows that RecOS consistently outperforms the other five methods as concurrency increases. Compared to TSMS and StreamRec, which utilizes Multistreams, RecOS can achieve up to a 68% speedup (WnD, concurrency 30) and a 63% speedup (WnD, concurrency 30), respectively. Both TSMS and StreamRec assign streams based on the topology order of computation graph. When faced with high concurrent queries, they encountered disordered scheduling. Besides, both of them adopt a naive tensor management strategy (as shown in Figure 5 (b)), which also introduced high overhead. As a result, both of them had degraded latency. The improvement of RecOS over TSSS and MSSS is not obvious at low concurrency ($\leq$ 8). However, the improvement increases as concurrency rises. This is because as concurrency increases, more ops need to be scheduled, thus a highly efficient scheduler can improve the inference latency. Besides, RecOS achieves a 46% improvement (DIN) at a concurrency of 30 compared to TSSS. The best improvement of RecOS over MSSS and Opara is about 44% and 53% for DIN respectively, at a concurrency of 30. Although both of them utilize multiple streams, they lacks in avoiding cache conflicts between embedding ops, thus leading to sub-optimal performance. What's more, the Tensor Manager adopted by RecOS reduces the overhead of synchronization between ops, which also makes RecOS better than Opara.

4.3 Performance under Online Traffic

Our experiments were conducted on RecOS and TSSS under simulated network traffic conditions. Specifically, we tailored the distribution of traffic to reflect the typical patterns encountered in business environments, focusing on three main types: pulse-type, bimodal, and unimodal traffic distributions.

The data depicted in these figures revealed two characteristics:

1. When the traffic volume is low, the performance of the TSSS is quite similar to that of RecOS. During these periods of low traffic, the latency difference between TSSS and RecOS is not significant.
2. When the traffic volume is high, RecOS demonstrates its effectiveness. It is evident that RecOS outperforms TSSS under conditions of high traffic volume. Furthermore, as seen in Figure 7, when traffic reaches its peak, the time saved by RecOS is maximized.

4.4 Performance Analysis

Better op schedule. The performance of RecOS can be attributed to several key factors:

1. **Efficient Stream Management** Unlike TSSS that uses only one stream or MSSS's simple stream assignment strategy, RecOS dynamically assigns ops to multiple streams based on comprehensive workload monitoring. This allows for better GPU resource utilization and reduced idle time.
2. **Cache Conflict Avoidance** RecOS's round-robin embedding launch algorithm minimizes cache conflicts, an

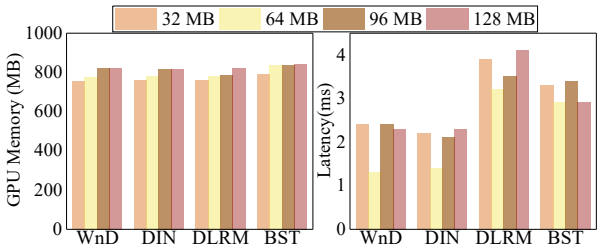

Figure 8: Comparison of Tensor Manager performance for all 4 models at different memory threshold. (a) GPU Memory Usage and (b) inference latency.

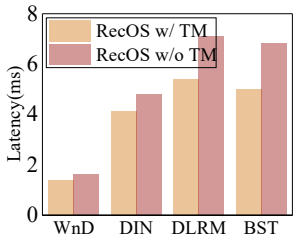

Figure 9: Comparison of RecOS with and without Tensor Manager.

issue overlooked by StreamRec and Opara. This is particularly important for RMs due to their mutable nature during inference.

3. **Topology-aware Scheduling** By considering both the $DepValue$ and topology of the computation graph, RecOS ensures critical ops are prioritized appropriately.

4. **Dynamic Workload Balancing** The system's workload monitor provides real-time information about stream status and resource utilization, enabling RecOS to make more informed scheduling decisions.

These optimizations work together to reduce inference latency, particularly during high-concurrency scenarios where the benefits become most apparent.

Tensor Management. By eliminating the post-processing step of tensor memory management, our tensor management reduces the overall latency. We first conducted experiments on all four models to investigate the impact of various memory threshold settings on system performance, as shown in Figure 8. As shown in Figure 8, under the threshold of 64 MB, the tensor manager achieves the best inference latency across all models. As the threshold increases, the frequency of tensor releases decreases, but the overhead associated with each release becomes larger. Additionally, the process tensor release also consumes resources (for example, event managers and thread pools), that are needed for op execution. The threshold of 64 MB achieves an optimal balance between release frequency and release overhead. Unlike inference latency, GPU Memory utilization increases as the memory threshold increases because more tensors are stored in the tensor manager's pool. However, it is noteworthy that even when the threshold reaches 128 MB, the GPU Memory Utilization only increases by about 100 MB, which is minimal compared to the overall GPU Memory capacity of 24 GB for A30.

Then we conducted experiments to investigate the impact of Tensor Management on RecOS at the threshold of 64 MB. Figure 9 shows that Tensor Manager improves inference latency of RecOS across all models. The comparative analysis conclusively shows that our tensor management' effectiveness while incurring only a marginal memory overhead.

5 Related Work

Recommendation Models Acceleration. Recently, many works have been proposed to accelerate DLRMs. Methods such as embedding cache on GPUs [Xie *et al.*, 2022; Wang *et al.*, 2022], op fusion [Pan *et al.*, 2023], near-memory processing [Ke *et al.*, 2020; Kwon *et al.*, 2019], and in-memory processing [Wang *et al.*, 2021] have been proposed to improve the embedding lookup, which are orthogonal to RecOS. StreamRec [Niu *et al.*, 2023] utilizes multiple streams to improve the embedding lookup op. However, all of them neglect the mutable nature of RM inference. To the best of our knowledge, RecOS is the first system that schedules RM inference based on RM's mutable nature.

GPU Concurrency. Many efforts have been proposed to increase GPU concurrency. Miriam [Zhao *et al.*, 2023] generates kernels with elastic resource occupancy and dynamically adjusts them when critical kernels arrive. Opara [Chen *et al.*, 2024] determines stream allocation and launch order based on computation graph topology and operation characteristics, generating CUDA Graphs to accelerate inference. Unlike these works, RecOS introduces a comprehensive scheduling mechanism that efficiently distributes RM operations across multiple streams, fully utilizing GPU resources while specifically addressing the unique mutable nature of recommendation models.

6 Conclusion and Future Work

We propose a novel system named RecOS, which improves latency in RM online service systems under concurrent queries. To tackle the cache conflicts caused by RM's mutable nature, RecOS proposes a round-robin embedding launch algorithm. The proposed system also takes advantage of the high parallelism topology of RM models and low-resource-consumption ops to achieve highly efficient parallel op launch. Furthermore, it utilizes a runtime op scheduler to process multiple concurrent queries across multiple stream to solve the disordered scheduling problem.

Experimental results with multiple models indicate that RecOS can enhance the overall system latency while incurring a minimal increase in memory occupancy. In the future, we plan to dive into the optimal configuration of RecOS for RM models.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grants (62471055, U23B2001, 62321001, 62101064, 62171057, 62201072), the Ministry of Education and China Mobile Joint Fund (MCM20200202, MCM20180101), the Fundamental Research Funds for the Central Universities (2024PTB-004). The research was also supported by Yerui Sun and Yuchen Xie from Meituan.

References

[Chen *et al.*, 2018] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[Chen *et al.*, 2019] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for e-commerce recommendation in alibaba, 2019.

[Chen *et al.*, 2024] Aodong Chen, Fei Xu, Li Han, Yuan Dong, Li Chen, Zhi Zhou, and Fangming Liu. Opara: Exploiting operator parallelism for expediting dnn inference on gpus, 2024.

[Cheng *et al.*, 2016] 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 *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery.

[Covington *et al.*, 2016] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. pages 191–198, Boston Massachusetts USA, September 2016. ACM.

[Ke *et al.*, 2020] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim 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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, Valencia, Spain, 2020. IEEE.

[Koren *et al.*, 2009] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, August 2009.

[Kwon *et al.*, 2019] 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, Columbus OH USA, 2019. ACM.

[Naumov *et al.*, 2019] 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, May 2019. arXiv:1906.00091 [cs].

[Niu *et al.*, 2023] Yuean Niu, Zhizhen Xu, Chen Xu, and Jiaqiang Wang. Accelerating Recommendation Inference via GPU Streams. volume 13943, pages 546–561. Springer Nature Switzerland, Cham, 2023.

[NVIDIA, 2024a] NVIDIA. Multiple stream. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, 2024.

[NVIDIA, 2024b] NVIDIA. Nvidia nsight systems. https://developer.nvidia.com/nsight-systems, 2024.

[NVIDIA, 2024c] NVIDIA. Nvidia triton inference server. https://github.com/triton-inference-server/server, 2024.

[Olston *et al.*, 2017a] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[Olston *et al.*, 2017b] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[Pan *et al.*, 2023] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, and Wei Lin. RECom: A Compiler Approach to Accelerate Recommendation Model Inference with Massive Embedding Columns. 2023.

[Wang *et al.*, 2021] Yitu Wang, Zhenhua Zhu, Fan Chen, Mingyuan Ma, Guohao Dai, Yu Wang, Hai Li, and Yiran Chen. Rerec: In-reram acceleration with access-aware mapping for personalized recommendation. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[Wang *et al.*, 2022] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. Merlin HugeCTR: GPU-accelerated recommender system training and inference. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 534–537, Seattle WA USA, 2022. ACM.

[Xie *et al.*, 2022] 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 *EuroSys '22: Seventeenth European Conference on Computer Systems*, pages 402–416, Rennes France, 2022. ACM.

[Zhao *et al.*, 2019] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. Recommending what video to watch next: a multi-task ranking system. pages 43–51, Copenhagen Denmark, September 2019. ACM.

[Zhao *et al.*, 2023] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. Miriam: Exploiting Elastic Kernels for Real-time Multi-DNN Inference on Edge GPU, 2023.

[Zhou *et al.*, 2018a] 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. pages 1059–1068, London United Kingdom, July 2018. ACM.

[Zhou *et al.*, 2018b] 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. pages 1059–1068, London United Kingdom, July 2018. ACM.

[Zhou *et al.*, 2019] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):5941–5948, July 2019.