# RecStream: Graph-aware Stream Management for Concurrent Recommendation Model Online Serving

**Shuxi Guo  Qi Qi  Haifeng Sun  Jianxin Liao**[†]
**Jingyu Wang**[†]
State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications, Beijing, China
{sx, qiqi8266, hfsun, wangjingyu}@bupt.edu.cn
jxlbupt@gmail.com

## Abstract

Recommendation Models (RMs) are crucial for predicting user preferences and enhancing personalized experiences on large-scale platforms. As the application of recommendation models grows, optimizing their online serving performance has become a significant challenge. However, current serving systems perform poorly under highly concurrent scenarios. To address this, we introduce RecStream, a system designed to optimize stream configurations based on model characteristics for handling high concurrency requests. We employ a hybrid Graph Neural Network architecture to determine the best configurations for various RMs. Experimental results demonstrate that RecStream achieves significant performance improvements, reducing latency by up to 74%.

## 1 Introduction

Recommendation Models are machine learning models used to predict users' preferences. An RM often consists of embedding lookup layers, which are memory-intensive parts, and several fully connected layers, which are compute-intensive parts. In recent years, companies like Google (Zhao et al., 2019; Covington et al., 2016), Alibaba (Zhou et al., 2018, 2019), and Netflix (Koren et al., 2009) have increasingly applied deep learning techniques to enhance the representation and prediction capabilities of RMs. RMs are critical for online services, particularly on large-scale platforms where personalized experiences drive user engagement and revenue. According to (Corinna Underwood, 2020), 75% of Netflix views and 60% of YouTube homepage clicks are driven by RMs. According to (Liu et al., 2022), Baidu processes billions of concurrent requests each day.

Given the significant role of recommendation systems, optimizing their online serving performance has become an important challenge. To handle such a massive volume of requests, production environments require an efficient online serving system which can 1) process massive concurrent requests within strict service level agreement (SLAs) (Liu et al., 2022; Jiang et al., 2021) to enhance user satisfaction and maximize revenue, and 2) make the best use of computational infrastructure to reduce unnecessary costs. However, current deep learning frameworks like Tensorflow (Abadi et al., 2015) and TorchRec (Ivchenko et al., 2022) are mainly focused on model training and lack serving efficiency. Machine learning compilers like TVM (Chen et al., 2018) improve inference latency by optimizing the computation graph topology and operator efficiency, but also lack online serving ability. Model serving frameworks like Tensorflow Serving (Olston et al., 2017) are proposed to meet online serving requirements, but they also perform poorly under high concurrency. We used Tensorflow Serving (Olston et al., 2017) to test

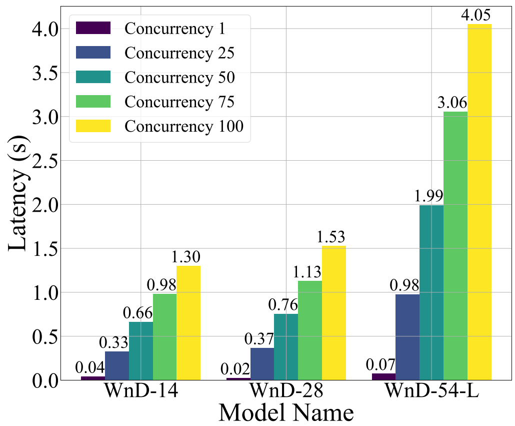



Figure 1: Online inference performance was evaluated at five levels of concurrency: 1, 25, 50, 75, 100. Results showed that at a concurrency level of 100, the inference latency can increase by as much as 57 times compared to the low concurrency scenario (WnD-54-L). The detailed numerical results are provided in Appendix A.

[†]Corresponding author.



*Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*, pages 817–826
January 19–24, 2025. 

several RMs under different concurrency scenarios and found that the inference latency increased drastically as concurrency increases. The increased latency primarily stems from the default CUDA stream scheduling mechanism, which processes operators in a First Come First Served (FCFS) manner, leading to resource contention. When processing concurrent requests, operators from different requests are sent to the same stream and cause contention for GPU resources. Although using multiple streams could alleviate this issue, the optimal stream configuration varies for different models due to the different model topologies and operator characteristics, which presents challenges for stream configuration of online serving.

To address these challenges, we propose **RecStream**, a system designed to find the optimal stream configurations for different models based on their model characteristics. It is difficult to determine the optimal stream configuration through simple rule-based methods because model topology, operator characteristics, and concurrency levels all impact latency. In recent years, Graph Neural Networks (GNNs) have gained popularity due to their powerful graph processing capabilities. To effectively consider both model characteristics and concurrency levels, we propose a heterogeneous graph neural network that integrates concurrency information into graph features for joint optimization. We conducted experiments on multiple production models under different concurrency levels, and the results demonstrate that our approach can achieve up to 75% performance improvement.

## 2 Related Work

**Deep learning serving systems** Many efforts have been made to build efficient serving systems systems (Fan et al., 2019; Liu et al., 2021, 2022; Gupta et al., 2020; Gujarati et al., 2020; Han et al., 2022; Ng et al., 2023; Strati et al., 2024). As a leading search engine company, Baidu proposed a series of DNN-based recommendation model serving systems (Liu et al., 2021; Fan et al., 2019; Liu et al., 2022), which handle massive requests efficiently. DeepRecSys (Gupta et al., 2020) proposes a recommendation serving scheduler to maximize throughput by considering the characteristics of online traffic patterns, model compute characteristics, and hardware systems. Clockwork (Gujarati et al., 2020) builds a fully distributed serving system by considering whether the GPU can meet the request deadlines, which can serve thousands of DNNs per server while achieving tight request-level service-level objectives (SLOs). REEF (Han et al., 2022) utilizes DNN kernel properties and employs a preemption scheme to better schedule between latency-critical and best-effort DNN inference tasks. Paella (Ng et al., 2023) enables software control of kernel execution order over the black-box GPU scheduler through a model compiler, local clients, and scheduler co-design. Orion (Strati et al., 2024) schedules operators by considering both their compute and memory requirements under multi-model concurrent serving scenarios. However, these existing optimization works on serving systems do not take the concurrency level into consideration, thus lack flexibility when deployed in online services.

**Machine Learning Compilers** In recent years, machine learning compilers have been widely proposed (Sabne, 2020; Chen et al., 2018; Pan et al., 2024; Zheng et al., 2023; Tillet et al., 2019; Zheng et al., 2022; NVIDIA, 2024a) due to their high efficiency and good portability. XLA (Sabne, 2020) and TVM (Chen et al., 2018) compile the machine learning computation graph into a series of fused computing kernels on a variety of devices, including CPUs, GPUs, and accelerators (e.g., FPGAs, ASICs). BladeDISC (Zheng et al., 2023) tackles the dynamic shape problem in ML models by shape information propagation and a compile-time and runtime combined code generation approach. Astitch (Zheng et al., 2022) uses a hierarchical data reuse technique and adaptive thread mapping to optimize memory-intensive ML computations. Recom (Pan et al., 2024) optimizes the heavy embedding computations in RMs by using a novel inter-subgraph parallelism-oriented fusion method to generate efficient code." Additionally, Triton (Tillet et al., 2019) was proposed to generate efficient GPU kernels for deep learning workloads. Our work is orthogonal to these compilation-related works and can be further accelerated with the proposed structured features and runtime modules after compilation optimization.

**Graph Neural Networks** Recently, GNNs have been widely used due to their ability to process data with graph structures. Graph Convolutional Networks (GCN) (Kipf and Welling, 2017) effectively aggregate features from a node's local neighborhood, making them particularly suitable for downstream tasks by extending convolution operations to graph structures using spectral graph theory. GraphSAGE (Hamilton et al., 2018) scales to large graphs and supports inductive learning, making it applicable to dynamic graphs and representation learning for unseen nodes by introducing a sampling and aggregation framework. Graph Autoencoders (GAE) (Hamilton et al., 2017) apply autoencoder architectures to graph data to capture latent representations of nodes. Recently, GNNs have also been applied to compiler. For example, (Brauckmann et al., 2020) uses GNNs instead of sequence models to capture the graph representations of code for learning compiler optimization tasks.

## 3 Method

In this section, we describe the details of our RecStream. We introduce the graph construction in Section 3.1, describe the network architecture in Section 3.2, and present the loss function in Section 3.3.

### 3.1 Graph Construction for GNN

**Graph Definition.** We formulate the computation graph as a directed graph $G = (V, E)$, where $V$ is the set of nodes and $E \subseteq V \times V$ is the set of edges. Each node $v \in V$ corresponds to an operator in the model, and each directed edge $(u, v) \in E$ represents a data dependency from operator $u$ to operator $v$. This structure captures the flow of computation and data within the model.

**Node Feature Construction.** Each node $v$ is associated with a feature vector $F_v \in \mathbb{R}^d$ that encapsulates essential attributes of the operator. The feature vector comprises three main components:

**Latency** ($l_v$): The average execution time of operator $v$ at different concurrency levels. This scalar value provides insight into the operator's performance characteristics.

**Operator Type** ($t_v$): A one-hot encoded vector representing the type of operator, where $t_v \in \{0, 1\}^K$ and $K$ is the total number of operator types. The $k$-th element of $t_v$ is set to 1 if operator $v$ is of type $k$, and 0 otherwise.

**Attribute Values** ($a_v$): A vector comprising both categorical and numerical attributes of the operator. Categorical attributes (e.g., data types) are one-hot encoded, while numerical attributes (e.g., tensor shapes, dimensions) are normalized to ensure consistent scaling.

The complete node feature vector is constructed by concatenating these components:

$$h_v = l_v \oplus t_v \oplus a_v \tag{1}$$

where $\oplus$ denotes vector concatenation.

### 3.2 Network Architecture

The architecture of RecStream is shown in Figure 2. With the graph $G$ and node features $\{F_v\}_{v \in V}$ defined, we employ a GNN to predict the optimal stream configuration.

**Graph Neural Network Layers.** We utilize two GCNs (Kipf and Welling, 2017) layers to process the graph. The GCN layers update each node's representation by aggregating information from its neighbors:

$$h_v^{(l+1)} = ReLU\left(W \cdot \frac{1}{|\mathcal{N}(v)|} \sum_{v' \in \mathcal{N}(v)} h_{v'}^{(l)}\right) \tag{2}$$

where $h_v^{(l+1)}$ represents the updated feature vector of node $v$ at layer $(l+1)$, $h_{v'}^{(l)}$ indicates the feature vector of neighboring nodes at layer $l$, $W$ is the weight matrix of the GCN, and $\mathcal{N}(v)$ denotes the set of neighbors of node $v$.

**Graph Embedding.** After the GCN layers, we obtain updated node representations $h^{(L)}$. We aggregate these representations into a single graph-level embedding $h_G$ using a global mean pooling operation:

$$h_G = \frac{1}{|V|} \sum_{v \in V} h_v^{(L)} \tag{3}$$

This embedding captures the overall structural and feature information of the computation graph.

**Concurrency Representation.** We represent the current concurrency level as a one-hot encoded vector $c \in \{0, 1\}^C$, where $C$ is the maximum concurrency level considered.

We concatenate the graph embedding $h_G$ with the concurrency vector $c$:

$$Z = h_G \oplus c \tag{4}$$

**Output Layer.** The combined vector $Z$ is passed through two fully connected layers with ReLU activation functions, followed by a final fully connected layer and a Softmax function to produce the probability distribution $y \in \mathbb{R}^S$ over possible stream configurations:

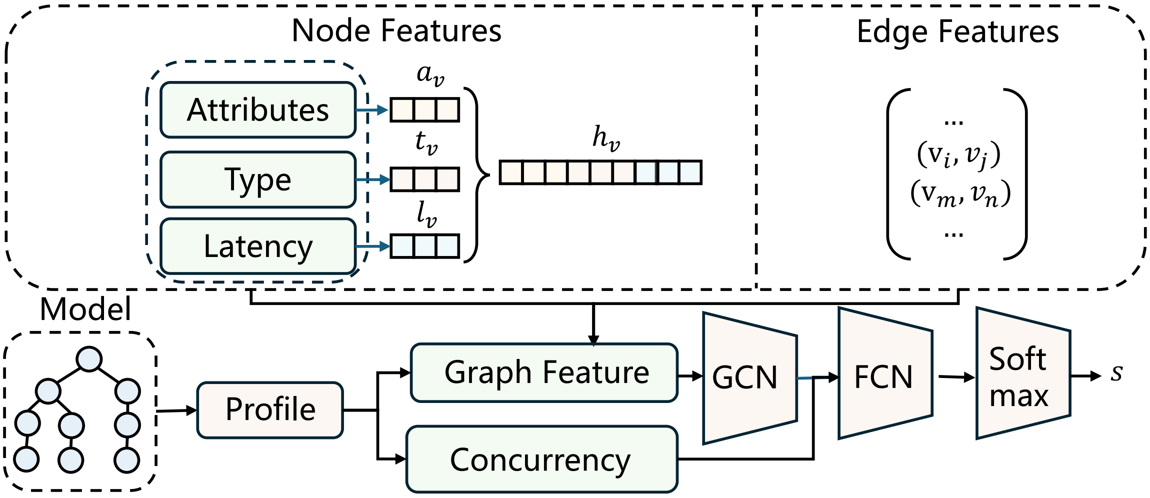



Figure 2: The architecture of our RecStream

$$y = \text{Softmax}(FCN(Z)) \tag{5}$$

where $S$ is the total number of stream options.

### 3.3 Loss Function

Our goal is to predict the optimal stream configuration under different concurrency levels. For each concurrency level, the ground truth is the number of streams that yield the best average latency performance. During training, this ground truth is transformed into a one-hot vector representation $Y \in 0, 1^S$, where $S$ is the total number of possible stream configurations. The model predicts a probability distribution over these configurations, denoted as $y$.

The standard cross-entropy loss is then used to compare the predicted distribution $y$ with the one-hot encoded ground truth. The loss function $L$ is defined as:

$$L = -\sum_{i=1}^{S} Y_i \log(y_i) \tag{6}$$

## 4 Experiments

In this section, we detail the experimental setup.

### 4.1 Experimental Setup

**Service Framework.** We implemented RecStream based upon DeepRec (Intelligence, 2023), an open-source recommendation model serving system designed for production-scale environments. Compared to the default TensorFlow Serving (Olston et al., 2017), DeepRec incorporates multi-stream and stream merging technologies to enhance online inference performance. These features enable more efficient utilization of GPU resources by allowing concurrent execution of multiple inference tasks and reducing kernel launch overhead.

**Hardware and Software Configuration.** All experiments were conducted on a server equipped with an Intel Xeon Platinum 8352Y CPU and an NVIDIA A30 GPU with 24 GB HBM2 memory, which is the same as our production environment setup. The system runs on CentOS with CUDA driver version 525 and CUDA Toolkit 12.0. All code was compiled using GCC 9.3.0 and `nvcc` with the `-O3` optimization flag to ensure performance.

**Models Evaluated.** We evaluated four real-world rms that are actively deployed in our online services. All these models are based on the Wide and Deep (WnD) architecture (Cheng et al., 2016), which is widely adopted in the recommendation systems domain due to its ability to handle both memorization and generalization by combining linear and nonlinear feature transformations.

The models, denoted as WnD-14, WnD-28, WnD-54-S, and WnD-54-L, were selected to represent a broad spectrum of recommendation model complexities and structures:

- **WnD-14** and **WnD-28** are lightweight models with lower computational demands (14 and 28 MFLOPs, respectively) and differ in the number of operators and features they process. They are representative of models used in scenarios where latency and resource constraints are critical, such as mobile applications or real-time recommendation systems.

- **WnD-54-S** and **WnD-54-L** are more complex models (both with 54 MFLoperators) but differ in their architectural designs. WnD-54-S has fewer operators (71 operators) with more complex computations per operator, representing models that perform intensive computations with deep feature interactions. WnD-54-L has more operators (101 operators) with simpler computations per operator, reflecting models that utilize wide architectures with extensive feature combinations.

These models cover a range of architectural patterns commonly found in recommendation systems, including variations in depth, width, and operator complexity. By evaluating RecStream on these diverse models, we aim to demonstrate its effectiveness across different types of RMs used in various real-world applications.

Table 1 summarizes their key characteristics. We utilized `tf.profiler` and NVIDIA Nsight Compute (NVIDIA, 2024b) for performance profiling and FLOPs computation.

These models, if not optimized, have a large number of small computational kernels, which can lead to significant kernel launch overhead on GPUs. To mitigate this, we applied optimizations using `TVM` (Chen et al., 2018), an open-source deep learning compiler stack that enhances performance by fusing kernels and reducing launch overhead. The FLOPs for each model were computed by summing the operations of both TensorFlow original operators and `TVM`-generated optimized operators.

**Data Collection.** To train our GNN-based model for stream configuration, we collected model performance data (i.e. mean latency) under different concurrency levels and stream configurations. Specifically, we conducted experiments at five levels of concurrency: 1, 8, 15, 22, and 30. At each concurrency level, a fixed number of clients continuously sent requests to the server to maintain the desired level of concurrency. It is noteworthy that the inference latency is defined as the duration of model computation excluding serialization, deserialization, and network transmission times. Finally, our dataset was composed of the model characteristics (e.g., model topology and operator characteristics) and latency under different combinations of concurrency and stream.

Table 1: Model Characteristics

| Model | FLOPS (MFLOPs) | #Ops |
|---|---|---|
| WnD-14 | 14 | 616 |
| WnD-28 | 28 | 179 |
| WnD-54-S | 54 | 71 |
| WnD-54-L | 54 | 101 |

**Training Details.** We implemented our GNN model using PyTorch Geometric (`PyG`) (Fey and Lenssen, 2019), a library specialized for graph neural networks. We employed the Adam optimizer (Kingma and Ba, 2017) with a learning rate of 0.001. To prevent overfitting, we applied a dropout rate of 0.5 after the GCN layers. The model was trained for 200 epochs with a batch size of 32. We split the dataset into training (80%) and test (20%) sets.

**Baselines.** We compared the performance of RecStream with the following baseline approaches:

1. **DeepRec-SS (DeepRec Single Stream)**: This baseline uses the DeepRec framework with a single CUDA stream for all inference tasks. This configuration is similar to the default setting of TensorFlow Serving (Olston et al., 2017), which also utilizes a single CUDA stream without concurrency optimization. Therefore, the performance of DeepRec-SS effectively represents that of TF-Serving, serving as a standard baseline without any concurrency optimization.

2. **DeepRec-Default (DeepRec Default Configuration)**: The default configuration of DeepRec, which utilizes a fixed number of four CUDA streams for inference. This setting is commonly used in production due to its balance between performance and resource utilization.

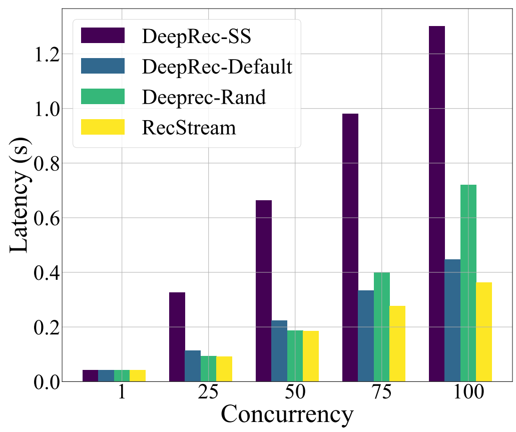
3. **DeepRec-Rand (DeepRec Random Configuration)**: In this baseline, we randomly assign stream configurations for different models and concurrency levels within a reasonable range.

## 5 Results

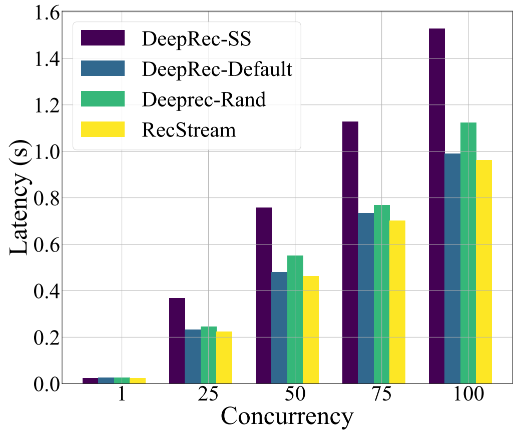
In this section, we present and analyze the performance of RecStream compared to the baselines.

### 5.1 Latency

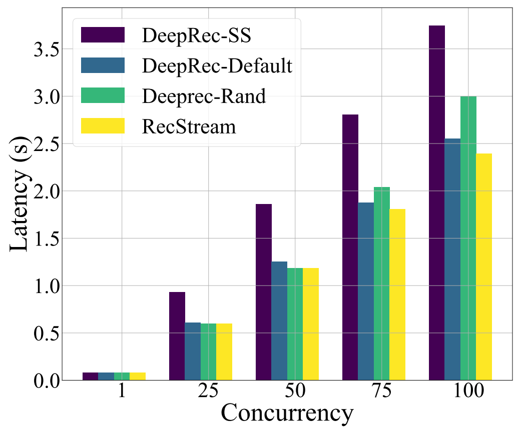
Figure 3 indicates that, as concurrency increases, nearly all schemes outperform DeepRec-SS config-



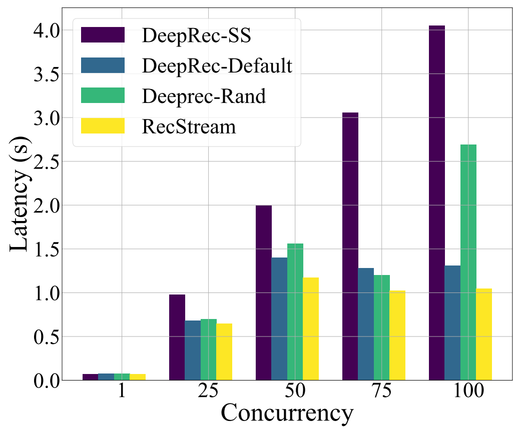

(a) WnD-14




(b) WnD-28




(c) WnD-54-S




(d) WnD-54-L

Figure 3: The mean latency of RecStream and other baselines under different levels of concurrency. The detailed numerical results are provided in Appendix A

urations, demonstrating the advantages of multi-stream. When compared to DeepRec-SS, RecStream achieves the best performance gain for model WnD-54-L at the concurrency level of 100. While RecStream does not outperform other multiple-stream baselines under single concurrency, its optimization gains relative to the other baselines increase significantly as concurrency increases. Compared to DeepRec-Default, RecStream maintains a consistent advantage. For model WnD-54-L, RecStream achieves a performance gain of nearly 74% compared to DeepRec-Default. It's noteworthy that, in some cases (e.g., WnD-54-L, concurrency 75), DeepRec-Rand performs better than DeepRec-Default, indicating that applying a fixed stream configuration across all models can yield suboptimal results. Overall, the experimental outcomes confirm that RecStream effectively selects the most suitable stream configurations for various models under different concurrency levels.

### 5.2 Ablation Study

**Concurrency** We found that as concurrency increases, the performance improvements of RecStream over `DeepRec-SS` and DeepRec-Default gradually increase. This is due to the network architecture in RecStream. With the help of GNN, RecStream can understand the model architecture and find optimal stream configurations under different concurrency levels.

**FLOPs** Additionally, we observed that model size significantly influences RecStream's performance. As the number of operators increases, RecStream's gain increases. For RecStream, the latency of WnD-54-L (nearly 1s) is far less than that of WnD-54-S (nearly 2.5s) at the concurrency level of 100. Although WnD-54-S and WnD-54-L have the same FLOPs, WnD-54-L has more operators. We argue that this is because as the number of operators increases, the contention between operators increases. By reducing contention, RecStream can achieve better performance.

### 5.3 Overhead Analysis

Here, we analyze the overhead of our proposed method.

**Offline Overhead**: The primary overhead of RecStream is during the offline training phase of the GNN Models. The training process is efficient and can be completed within a few hours. Moreover, models in production environments do not change frequently, so the GNN model does not require frequent retraining.

**Online Overhead**: In the online serving phase, RecStream introduces negligible overhead. Once trained, the GNN model is lightweight and can quickly predict the optimal stream configuration based on the model's characteristics and the current concurrency level. This prediction is performed infrequently (e.g., when the concurrency level changes significantly) and does not impact the inference latency.

Despite the need for offline training, the proposed method is worthwhile. In recommendation systems, latency is a critical factor impacting user experience and system efficiency. Even small reductions in latency can lead to substantial cost savings and increased revenue.

## 6 Conclusion

In this work, we present RecStream, a hybrid network architecture that determines optimal online serving configurations based on model characteristics and concurrency levels. By utilizing GCNs, RecStream can find the best stream configuration for various RMs under different levels of concurrency. RecStream outperforms other simple, fixed stream configuration methods that use the same settings for all RMs.

## Acknowledgments

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)

## References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, New York, NY, USA. Association for Computing Machinery.

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. 2018. 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. USENIX Association.

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. 2016. 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. Association for Computing Machinery.

Corinna Underwood. 2020. Use cases of recommendation systems in business – current applications and methods. `https://emerj.com/ai-sector-overviews/use-cases-recommendation-systems/`.

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

Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. 2019. MOBIUS: Towards the Next Generation of Query-Ad Matching in Baidu's Sponsored Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge*

*Discovery & Data Mining*, pages 2509–2517, Anchorage AK USA. ACM.

Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with pytorch geometric. *Preprint*, arXiv:1903.02428.

Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association.

Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, Valencia, Spain. IEEE.

William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216.

William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive representation learning on large graphs. *Preprint*, arXiv:1706.02216.

Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA. USENIX Association.

Alibaba Cloud Data Intelligence. 2023. Deeprec: A training and inference engine for sparse models in large-scale scenarios.

Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. 2022. Torchrec: a pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, page 482–483, New York, NY, USA. Association for Computing Machinery.

Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3097–3105, Virtual Event Singapore. ACM.

Diederik P. Kingma and Jimmy Ba. 2017. Adam: A method for stochastic optimization. *Preprint*, arXiv:1412.6980.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. *Preprint*, arXiv:1609.02907.

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

Hao Liu, Qian Gao, Jiang Li, Xiaochao Liao, Hao Xiong, Guangxing Chen, Wenlin Wang, Guobao Yang, Zhiwei Zha, Daxiang Dong, Dejing Dou, and Haoyi Xiong. 2021. Jizhi: A fast and cost-effective model-as-a-service system for web-scale online inference at baidu. *Preprint*, arXiv:2106.01674.

Hao Liu, Qian Gao, Xiaochao Liao, Guangxing Chen, Hao Xiong, Silin Ren, Guobao Yang, and Zhiwei Zha. 2022. Lion: A GPU-Accelerated Online Serving System for Web-Scale Recommendation at Baidu. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3388–3397, Washington DC USA. ACM.

Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 595–610, New York, NY, USA. Association for Computing Machinery.

NVIDIA. 2024a. Getting started with cuda graphs. `https://developer.nvidia.com/blog/cuda-graphs/`.

NVIDIA. 2024b. Nvidia nsight compute. `https://docs.nvidia.com/nsight-compute/NsightComputeCli/`.

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

Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. 2024. 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*, ASPLOS '23, page 268–286, New York, NY, USA. Association for Computing Machinery.

Amit Sabne. 2020. Xla : Compiling machine learning for peak performance.

Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1075–1092, New York, NY, USA. Association for Computing Machinery.

Philippe Tillet, H. T. Kung, and David Cox. 2019. 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*, MAPL 2019, page 10–19, New York, NY, USA. Association for Computing Machinery.

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

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

Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 359–373, New York, NY, USA. Association for Computing Machinery.

Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep Interest Evolution Network for Click-Through Rate Prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):5941–5948.

Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. pages 1059–1068, London United Kingdom. ACM.

## A Detailed Numerical Results

In this appendix, we provide the detailed mean latency results for each model under varying concurrency levels and different methods.

| Concurrency | DeepRec-SS | DeepRec-Default | DeepRec-Rand | RecStream |
|---|---|---|---|---|
| 1 | 0.04 | 0.04 | 0.04 | 0.04 |
| 25 | 0.33 | 0.11 | 0.18 | 0.09 |
| 50 | 0.66 | 0.22 | 0.19 | 0.19 |
| 75 | 0.98 | 0.33 | 0.40 | 0.28 |
| 100 | 1.30 | 0.45 | 0.53 | 0.36 |

Table 2: Mean Latency (in seconds) for WnD-14 under Different Concurrency Levels

| Concurrency | DeepRec-SS | DeepRec-Default | DeepRec-Rand | RecStream |
|---|---|---|---|---|
| 1 | 0.02 | 0.03 | 0.03 | 0.02 |
| 25 | 0.37 | 0.23 | 0.23 | 0.22 |
| 50 | 0.76 | 0.48 | 0.55 | 0.46 |
| 75 | 1.13 | 0.73 | 0.77 | 0.70 |
| 100 | 1.53 | 0.99 | 1.12 | 0.96 |

Table 3: Mean Latency (in seconds) for WnD-28 under Different Concurrency Levels

| Concurrency | DeepRec-SS | DeepRec-Default | DeepRec-Rand | RecStream |
|---|---|---|---|---|
| 1 | 0.08 | 0.08 | 0.08 | 0.08 |
| 25 | 0.93 | 0.61 | 0.73 | 0.60 |
| 50 | 1.86 | 1.25 | 1.28 | 1.18 |
| 75 | 2.81 | 1.88 | 2.04 | 1.81 |
| 100 | 3.75 | 2.56 | 3.00 | 2.39 |

Table 4: Mean Latency (in seconds) for WnD-54-S under Different Concurrency Levels

| Concurrency | DeepRec-SS | DeepRec-Default | DeepRec-Rand | RecStream |
|---|---|---|---|---|
| 1 | 0.07 | 0.08 | 0.07 | 0.07 |
| 25 | 0.98 | 0.68 | 0.70 | 0.65 |
| 50 | 1.99 | 1.40 | 1.56 | 1.18 |
| 75 | 3.06 | 1.28 | 2.20 | 1.03 |
| 100 | 4.05 | 1.31 | 2.88 | 1.05 |

Table 5: Mean Latency (in seconds) for WnD-54-L under Different Concurrency Levels