Reducing the End-to-End Latency of DNN-based Recommendation Systems in GPU Pools

*Guangqiang Luan[1], *Pu Pang[1], *Quan Chen, *Chen Chen, ◇Guoyao Xu, ◇Chi Zhang
◇Yanyi Zi, ◇Yinghao Yu, ◇Guodong Yang, ◇Liping Zhang, *Minyi Guo
*Department of Computer Science and Engineering, Shanghai Jiao Tong University
◇Alibaba Group
{luan-gq, chen-chen}@sjtu.edu.cn, {pangpu, chen-quan, guo-my}@cs.sjtu.edu.cn
{yao.xgy, yujing.zc, ziyanyi.zyy, yinghao.yyh, luren.ygd, liping.z}@alibaba-inc.com

***Abstract*—While intelligent applications (e.g., recommendation systems) prefer different CPU-GPU ratios, GPU pooling technique that decouples the GPU and CPU resources yields substantial flexibility when serving diverse applications. With such architecture, DNN-based recommendation services often offload the compute-intensive neural network layers to the remote GPU pool for high resource utilization. However, such a paradigm results in the long end-to-end latency due to two causes: 1) the intermediate data is copied for multiple times during the entire process in current GPU pooling practices, incurring heavy overheads; 2) the content transferred to the GPU pool involves multiple small tensors, suffering from poor bandwidth efficiency. To solve these problems, we design Zero, a runtime system that incorporates a zero-copy transmission mechanism as well as a dynamic tensor merging policy. The zero-copy transmission mechanism unifies memory management across the inference framework and the RPC framework, accompanied by an elaborated serialization protocol to fully eliminate redundant data copying. Meanwhile, the tensor merging policy deliberately organizes small tensors into larger data blocks, so as to transfer them with higher efficiency. Experimental results show that, compared with prior work, Zero reduces the latency of typical recommendation models by up to 15.1% (10.1% on average).**

***Index Terms*—Recommendation Model, GPU Pool, Zero-copy Serialization**

I. Introduction

Recommendation systems are critical to the service profit of commercial companies, and they widely use deep neural networks (DNN) as the backbone. For instance, in Meta [1], up to 79% of the DNN inference queries are generated by the recommendation systems, other big players—like Google [2], Amazon [3] and Alibaba [4]—also have similar characteristics. In particular, the typical DNN models adopted in recommendation systems (e.g., BST [5], DIEN [4], DSSM [6], and ESMM [7]) all comprise embedding layers followed by the computational layers. Regarding the resource preferences of those layers, the embedding layers usually require large memory space to store the embedding tables, yet the computational layers require powerful computing units for matrix operations [8]. Given such resource preferences, for a typical DNN model in recommendation systems, the embedding layers are often processed on CPUs (with abundant memory resources), whereas the computational layers are processed on GPUs (to attain short response latency) [4], [9]–[12].

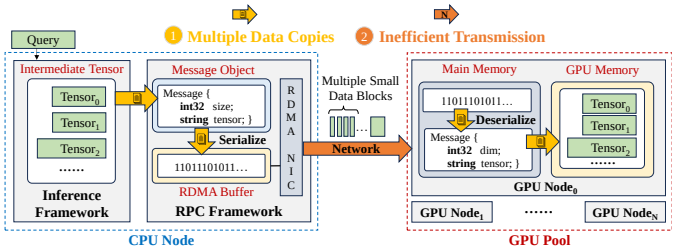


Fig. 1: An example recommendation system in GPU pools.

Cutting-edge computing servers in production clusters are equipped with a certain number of GPUs and CPU cores [13]–[15], yet such a fixed CPU-GPU resource ratio is inflexible for runtime inference workloads in recommendation systems. For example, on a server with two Intel Xeon Platinum 8369B CPUs and 8 A100 GPUs [16], our measurements show that the CPU utilization is already 60% when merely using a single A100 GPU for inference queries in our production.In that case, the other 7 GPUs cannot be fully utilized due to resource contention on CPUs, thus leading to low throughput.

To support flexible allocation of heterogeneous resources, a straightforward method is to build a separate GPU pool and connect it to traditional CPU servers using the fast network. The GPU pool offers substantial flexibility when hosting diverse applications with different CPU-GPU demand ratios. With such architecture, we could slice the recommendation model—running the memory-intensive part (e.g., embedding layers) on the local CPU server using traditional DNN inference framework, while offloading the GPU-intensive part (e.g., the remaining computational layers) to the remote GPU pool. The two parts communicate through Remote Procedure Call (RPC) framework with serialization library (e.g., Protobuf [17]). The actual data transmission between the CPU server and GPU pool can be done using RDMA (Remote Direct Memory Access) [18] protocol, like RoCE [19]—the industry standard Ethernet-based RDMA solution.

We further show an industrial example of the GPU-pool-based recommendation system with Figure 1. In order to

[1]Both authors contributed equally to this work.

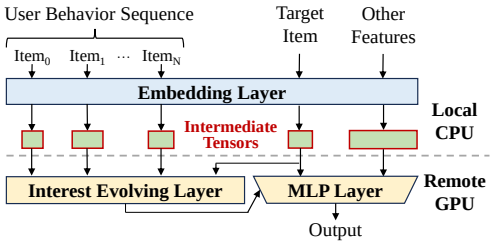


Fig. 2: Slicing of an e-commerce recommendation model.

offload the GPU-intensive part to the GPU pool, the intermediate tensors generated by the embedding layer (affiliated with metadata like tensor size) are serialized and encoded into an RPC message. The serialized message is built into the RDMA buffer to transfer with RDMA [20], and then sent to a GPU node based on the RPC scheduling policy (e.g., to a GPU node with the shortest queue [21]). The GPU node deserializes the received data, performs the following computation, and returns the recommendation results.

Although the above mechanism is easy for production usage, it would however result in *long latency* of the recommendation query, and there are two reasons for that.

First, serialization and deserialization are time-consuming due to multiple redundant data copying processes. As shown in Figure 1, in preparation for RPC transmission, the tensor is copied from the memory space of the inference framework to generate a message object in the RPC framework. A serialization library (e.g., Protobuf) then encodes the message object into a binary data stream for the transmission [22]. The serialization also introduces a data copying because the memory layout of the message object is different from the binary encoding format (such data copying also exists symmetrically in the deserialization phase). In particular, a recent work, Cornflakes [23], uses NIC Scatter-Gather (SG) to eliminate the data copying during serialization and deserialization, but it still requires data copying in generating the message object: NIC SG can only send data located in RDMA buffer, yet inference frameworks like TensorFlow and PyTorch store tensors out of the RDMA buffer. To summarize, redundant data copying in RPC transmissions brings non-negligible time overheads.

Second, transferring multiple small tensors to the pool is inefficient. As shown in Figure 2, in an e-commerce recommendation model [4], the embedding layer takes the user behavior sequence, the target item, and many other features as the inputs. These inputs are embedded as many different intermediate tensors and transferred to the GPU pool. The sizes of intermediate tensors to be transferred to the GPU pool are usually small. In our production recommendation model, there are more than one hundred intermediate tensors generated in a recommendation query while about 60% of them are smaller than 1KB. Similar distribution can be found in general recommendation models. However, when the RDMA NIC continuously transfers multiple small data blocks, the transmission cost may be higher than merging them into a new large data block and then transferring the new block. For example, it takes 10.9μs to consecutively transfer two 1KB data blocks, whereas transferring a merged version of one 2KB-block only takes 5.6μs.

We note that the *isolated design of the DNN inference framework and the RPC framework* is a chief culprit for the inefficiencies described above. With the isolated design, the two frameworks manage their data and memory space separately. In contrast, with framework co-design, it is possible for the inference framework to directly allocate the memory for the intermediate tensor in the memory region that can be directly reached by the RPC framework. In this way, all the tensors can be transferred to the GPU pool without being redundantly copied, and the small tensors can be organized into large data blocks during transmission. Although merging tensors shows the advantage, inappropriate merging may lead to low transmission efficiency on the contrary (detailed later in Figure 5). This is because merging too many or too large data blocks brings non-negligible extra cost.

We therefore propose **Zero**, a system that co-designs the inference and RPC framework. Zero comprises a *zero-copy transmission mechanism* and a *dynamic tensor merging policy*. The zero-copy transmission mechanism introduces unified memory management across the inference framework and RPC framework to enable transferring tensors with RDMA directly. An elaborated serialization protocol is provided in the mechanism to prevent tensors from being copied repeatedly in RPC transmission, and send metadata and tensors to the appropriate memory location of the GPU node in the pool separately. Meanwhile, we also design a tensor merging policy, which can identify which tensors should be merged for transferring according to the distribution of tensor sizes, with the objective of achieving higher transmission efficiency.

While Zero has been deployed in our production GPU pools, we conduct experiments on our local machines for the purpose of test flexibility. Our experimental results show that: 1) compared with Protobuf [17], Zero averagely reduces the end-to-end latency by 18.5% (up to 32.0%), and 2) compared with Cornflakes [23], Zero averagely reduces the end-to-end latency by 10.1% (up to 15.1%).

To summarize, this paper makes three main contributions:

- **A comprehensive analysis of the problems of serving recommendation systems with GPU pools.** We find the gap between inference framework and RPC framework hurts the latency of recommendation models.
- **An effective solution attaining short service latency of GPU-pool-based recommendation systems**. It avoids redundant data copying and also accelerates data transmission with deliberate tensor merging.
- **Product-level implementation and justified performance in production**. We have implemented Zero using TensorFlow [24] (as the inference framework), Protobuf (as the serialization library) as well as our in-house RPC framework (similar to eRPC [25]). Moreover, we have deployed Zero in production environment and verified its performance at a large scale.

II. Related Work

Optimization for Inference. For general DNN models, some studies focus on adaptively slicing the model for efficient inference [26], [27]. While Zero slices the recommendation models in a fixed pattern like prior work [4], [12]—running embedding layer on CPU and other layers on GPU, Zero can work with any slicing method because the design of Zero does not rely on any assumption about the slicing. Other studies focus on auto-tuning the batch size and GPU type during the inference [28], [29]. Since adjusting the configurations does not change the communication way between CPU and GPU nodes, Zero is orthogonal to these studies. Aiming at recommendation models, many prior studies focus on reducing the cost of the embedding lookup. For example, partitioning the embedding table [11], optimizing the embedding cache [30], and speeding up the processing of embedding layers by prefetching or in-storage computing [10], [31]. Instead of changing the processing of embedding layers, Zero optimizes the transmission of intermediate tensors between embedding layers and subsequent layers.

GPU Pooling. When remote GPU computing is required, there have been two kinds of studies: API Remoting [32], [33] and customized API for devices [34]. These studies require modifications on hardware or driver; without such modifications, this work splits the computation graph from the application layer and executes remote GPU inference through RPC calls (similar to eRPC [35]). Because naively using RPC communication leads to high communication overhead, this work reduces the overhead through inference framework and RPC framework co-design.

Acceleration for Serialization. There have been some serialization library [23], [36], [37] designed to reduce data copying. Flatbuffers [36] eliminates the data copying during deserialization with the cost of longer serialized data compared with Protobuf. Prior work [23], [38] shows that Flatbuffers and Protobuf have similar performance. Cornflakes [23] relies on the Scatter-Gather capability of the NIC to eliminate the data copying, but it still requires data copying when generating the message object. A serialization-free runtime called Naos [37] can write Java objects directly to the receiver. But Naos can only be used in Java runtime.

Merging Data Blocks in Transmission. Prior studies note that merging some small data blocks to transfer together can be better than transferring them separately [23], [39], [40]. They tend to use a static threshold to guide which data blocks should be merged together (e.g., zIO [40] merges data blocks whose sizes are not larger than 16KB). Our investigation shows that a static threshold is not suitable for different applications and workloads. When GPU pool is shared by different applications and workloads, Zero can dynamically set appropriate merge thresholds for them.

III. Motivation

A current GPU node often has multiple GPUs. For instance, Google, Meta, and Microsoft tend to use nodes with eight A100 GPUs [41]–[43]. With such stand-alone nodes, the CPU-GPU resource ratio is fixed, while a datacenter often runs multiple tasks that often have different optimal CPU-GPU ratios. For instance, a training task often prefers more GPUs on a single CPU node for quick GPU communication, and an inference task often prefers fewer GPUs on a node. In this case, it is not practical to physically move the GPU cards among CPU nodes, while the tasks may change frequently. To this end, this section seeks to answer two questions:

- Whether current node with multiple GPUs can efficiently host workloads like recommendation models that also have high CPU requirement? If not, can we use GPU pool to improve the GPU efficiency?
- What are the causes that result in the long latency of recommendation queries with GPU pool architecture?

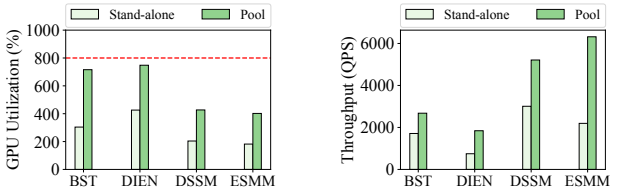
A. Advantages of GPU Pool

In addition to CPU, GPU is used in recommendation model inference [4], [12] because it is cost-effective. To illustrate, we conduct a stress test with one 64-logical-core CPU (Intel Xeon Platinum 8369B) by submitting a saturating number of inference queries for the recommendation model BST [5]. The batching technique [44] is enabled to improve the hardware utilization. We observe that the peak throughput is 3,093 QPS and the average latency is 10.9ms (the batch size is 4; increasing it does not improve throughput but increases the latency). In contrast, when using one A100 GPU (with a batch size of 128), the peak throughput reaches 18,560 QPS and the average latency is 9.6ms. This implies that six 64-core CPUs are required to achieve the same throughput as one GPU. According to the on-demand price on AWS [45], renting six 64-core CPU instances costs 16.32 USD per hour while renting an A100 GPU instance costs 4.09 USD per hour. This means that using the GPU saves 75% of the cost.

As GPU nodes in today's datacenters are often equipped with multiple GPUs, we first investigate the efficiency of a stand-alone node with multiple GPUs when serving recommendation models. We use four recommendation models (BST [5], DIEN [4], DSSM [6] and ESMM [7]) as the benchmarks. The benchmarks contain sequence features that reflect users' historical behaviors (sequence features are usually contained in production models). Similar to previous work [8], we add more feature columns and sequence features in the input datasets to reveal the case of large-scale recommendation models. We run the experiment on a node with two Intel Xeon Platinum 8369B CPUs. Besides, there are eight A100 GPUs on the node. Detailed experimental platform configuration is shown in Table I.

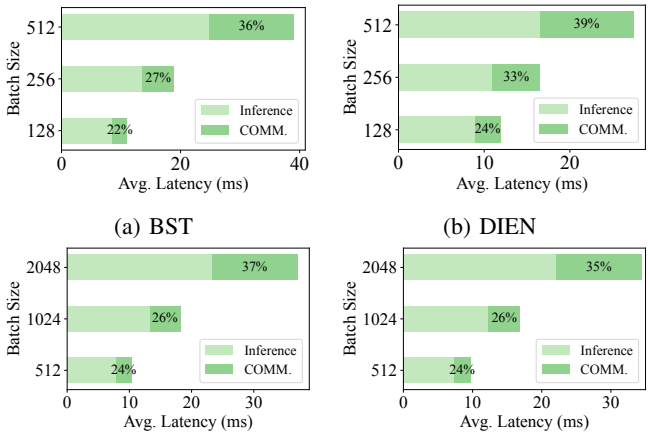
In the experiment, we run the embedding layer on the CPUs of the node and run the remaining layers on the GPUs. The batch sizes of BST and DIEN are set to 512 while the batch sizes of DSSM and ESMM are set to 2048. Using larger batch sizes will not improve the GPU utilization in the experiment of the stand-alone node.

The "Stand-alone" bar in Figure 3(a) shows the GPU utilization when the benchmarks consume all CPU cycles—



(a) GPU Utilization (b) Processing Throughput

Fig. 3: GPU utilization and throughput of widely-used recommendation models.



(a) BST (b) DIEN

(c) DSSM (d) ESMM

Fig. 4: The latency breakdown of four recommendation models deployed in the GPU pool.

the utilization of each CPU core of the node reaches 100%. In the figure, the GPU utilization will be 800% when all GPUs of the node are fully used because there are eight GPUs in total. As we can see, for the four benchmarks, the GPU utilization ranges from 182% to 426%. The GPUs are wasted, due to the lack of CPU resources to process the embedding layers of the recommendation model.

To improve the GPU utilization, a straightforward idea is building GPU pools, and letting normal CPU nodes collaborate with the nodes with multiple GPUs. GPU pooling allows us to freely tune the CPU-GPU ratio according to the requirements of workloads. Figure 1 shows the way that a GPU pool serves a recommendation model.

We then conduct an experiment to verify if pooling can improve the GPU efficiency. We put the above stand-alone GPU node (with eight A100 GPUs) in the pool and connect three CPU nodes (each CPU node is equipped with two Intel Xeon 8396B CPUs) to this GPU node, while running the four benchmarks respectively. The "Pool" bar in Figure 3 shows the GPU utilization and throughput of each recommendation model. As observed in the figure, pooling improves the GPU utilization: as there are more available CPU resources, those eight A100 GPUs can achieve higher utilization, and the throughput (the number of completed queries within a second) of the recommendation model increases at the same time.

B. Response Latency with GPU Pool

In this experiment, we analyze the latency of recommendation models with the GPU pool. We enable Multiple Instance GPU (MIG) [46] for GPUs to simulate real-system scenarios. MIG is used to partition a GPU into several parts for efficient sharing. Without loss of generality, we partition an A100 GPU into three MIG instances: 4g.40gb, 2g.20gb, and 1g.10gb. Given that an A100 GPU has 98 streaming multiprocessors (SMs) with MIG enabled, "2g.20gb" represents the GPU instance that has $98 \times (2/7) = 28$ SMs, and 20GB global memory. Default Protobuf [17] is used as the serialization library. The queries are routed to the three GPU instances using the JSQ [21] policy.

Figure 4 shows the average end-to-end latency (as well as the breakdown) of each benchmark. The end-to-end latency refers to the time it takes for a query to be submitted to the CPU node and to obtain the inference result. In the figure, the "Inference" part of the latency represents the computation time taken on CPU and GPU, while the other part is the communication overhead between the CPU node and GPU pool (denoted as COMM.).

We observe that the communication overhead accounts for a non-negligible proportion of the end-to-end latency, and that proportion increases as the batch size gets larger. For example, in DIEN, the communication overhead takes 24% of the latency when the batch size is 128, and the proportion increases to 39% when the batch size increases to 512.

The relative portion of communication increases as the batch size increases, because the benchmarks with small batch sizes do not fully utilize the computational resources. In this case, the computation time increase sub-linearly. For example, when the batch size of DIEN in Figure 4 is 128, 256, and 512, the computation time is 9.0ms, 11.0ms, and 16.5ms (increase sub-linearly). On the contrary, the communication time is 2.9ms, 5.5ms, and 11.0ms (increase near-linearly).

C. Root Causes of High Communication Overhead

We identify two main reasons that cause the high communication overhead.

Multiple Redundant Data Copies. As already shown in Figure 1, there exist four data copying operations when serializing and deserializing the intermediate tensors. It is possible to eliminate those data copying operations, if the RPC

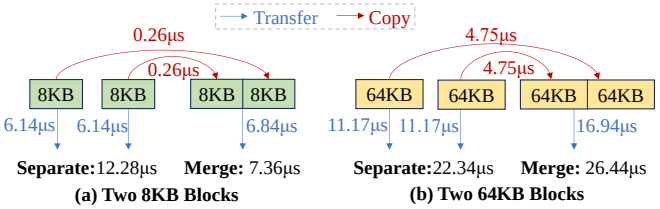


Fig. 5: The cost of transferring two data blocks separately and transferring the merged data block

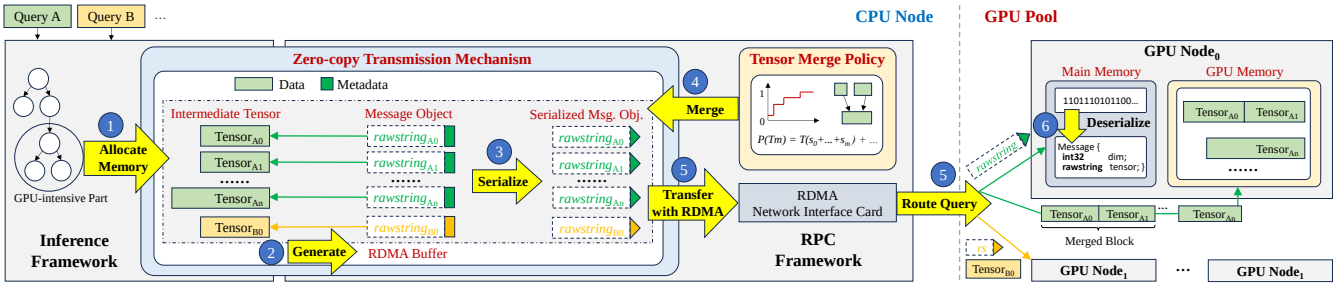


Fig. 6: Design overview of Zero, which includes two components—a transmission mechanism and a tensor merging policy.

framework is able to directly access the intermediate tensors in the inference framework and the serialization protocol frees the tensors from being copied in the RPC transmission. Such a zero-copy mechanism requires the co-design of the inference framework and the RPC framework.

Inefficient Transmission of Small Tensors. We find that there are many small intermediate tensors to be transferred when deploying recommendation models in the GPU pool. For example, 60% intermediate tensors in DIEN are not larger than 10KB. However, it is inefficient to separately transfer multiple small data blocks with RDMA. As shown in Figure 5(a), separately transferring two 8KB data blocks takes 12.28μs while transferring them after merging into a 16KB block only takes 7.36μs. Although merging data blocks incurs extra cost—it requires copying the to-be-merged data blocks to a new memory space and placing them adjacently, the cost is negligible when the data block is small. However, when there are too many blocks to be merged or the data block size is relatively large, the extra cost cannot be ignored, so separately transferring data blocks will be a better choice instead, as shown in Figure 5(b). A policy is required to identify which tensors should be merged to maximize transmission efficiency.

While some prior work [23], [36], [39], [40] can reduce the communication overhead to some extent, they are not efficient for AI framework-based GPU pool. For example, Cornflakes [23] uses NIC Scatter-Gather to eliminate the data copying during serialization and deserialization, but the data copying in generating the message object cannot be eliminated. This is because Scatter-Gather can only send data located in the RDMA buffer, yet the inference framework TensorFlow as well as PyTorch stores tensors out of the RDMA buffer. Besides, it cannot automatically send the intermediate tensor and the metadata to the appropriate memory location of the GPU node, resulting in another data copying process. We have a detailed comparison in Section VI.

IV. Design of Zero

In this section, we present an overview of Zero, followed by the detailed design of the zero-copy transmission mechanism and the dynamic tensor merging policy. Besides, we also discuss the way to build effective GPU pools based on the design and findings of Zero.

A. Overview

We therefore propose **Zero**, a system that co-designs the inference framework and RPC framework. Zero comprises a *zero-copy transmission mechanism* and a *dynamic tensor merging policy*. The zero-copy transmission eliminates all the data copies in RPC transmission. The tensor merging policy properly merges small tensors to make them be transferred efficiently. We use Figure 6 to explain how Zero works. When an inference query is submitted, we run the embedding layer on CPU node and offload the remaining computational layers to the remote GPU node [4], [11], [12].

① The zero-copy transmission mechanism introduces unified memory management across the inference framework and the RPC framework. By providing a shared memory space, the inference framework can use the unified memory allocator to directly allocate memory and store intermediate tensors into the RDMA buffer of the RPC framework. ② Zero generates the message object from the tensors. Instead of storing tensors in the message object, we introduce a new data type *rawstring* (it contains the pointer that points to the tensor) in the message object, so that the copy of tensors could be eliminated. ③ The generated object is serialized into a binary data stream (serialized message object). Since the message object does not include actual tensors, tensors will not be serialized (also not be copied) during the serialization.

④ Before transferring all tensors, the tensor merging policy merges some tensors according to the distribution of tensor sizes. ⑤ After that, the intermediate data is transferred to the remote GPU node. While there have been many routing policies [21], [47], [48], we route queries using the JSQ [21] policy that is widely used [49]–[51] in routing queries across multiple servers, based on the queue length of queries on each GPU node. During data transmission, the RPC framework will transfer the serialized message object to the main memory of the GPU node and transfer the corresponding tensor to the GPU global memory of the GPU node. ⑥ The GPU node receives the serialized object and deserializes it; there is no data copy since the serialized object does not include tensors.

Generally, the GPU pool hosts multiple models. It is usually not possible to load all models on every GPU due to the limitations of GPU memory. Models can be pre-loaded on some GPUs rather than all GPUs in the pool according to historical

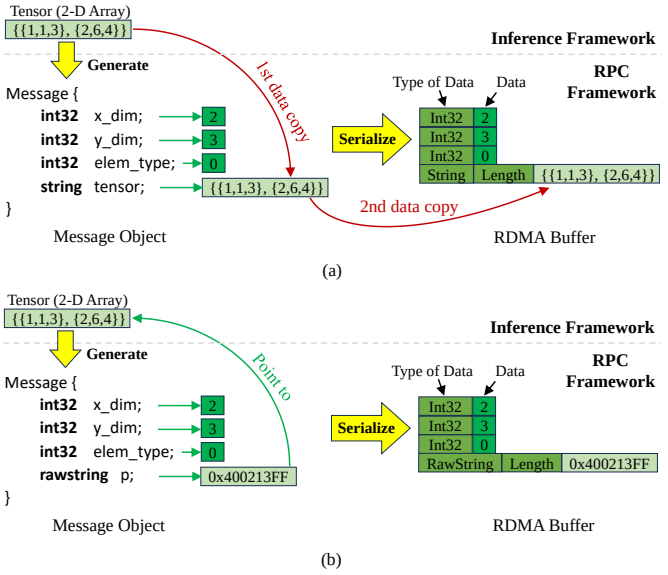


Fig. 7: (a) The original serialization process. (b) Our proposed serialization process.

statistics (model occupancy, traffic, etc.). Besides, NVIDIA Unified Virtual Memory (UVM) [52] technique, which supports GPU memory over-subscription, can be merged in Zero to achieve on-demand model loading.

Note that, the design of Zero is scalable, because one GPU instance [46] is allocated to a single service, each GPU instance uses a private memory pool that contains the buffer communicated with the CPU node and dynamically scales.

B. Zero-copy Transmission Mechanism

This subsection describes the way to eliminate redundant data copies in the RPC communication between the CPU node and the GPU pool.

Unified Memory Management. A main reason for redundant data copies is that the inference framework and the RPC framework manage their memory space separately. To this end, we propose to unify the memory management across the two frameworks to provide a shared memory space.

The data transferred with RDMA should be located in the registered memory region [20] (RDMA buffer). However, popular inference frameworks use their own memory allocators to allocate memory (e.g., *BFCAllocator* [53] in TensorFlow and *DefaultCPUAllocator* [54] in PyTorch) and store data outside the RDMA buffer. If we want to transfer this data, we have to copy the data to the buffer. To this end, we design a unified memory allocator that can allocate RDMA-registered memory. Using this allocator, the inference framework can directly store tensors into the RDMA buffer, and the serialization library can build the serialized data into the RDMA buffer as well. As a result, this data can be directly transferred with RDMA without copying.

The size of RDMA buffer should be carefully maintained because it continuously occupies a memory region while this memory region cannot be used by other applications. Hence,

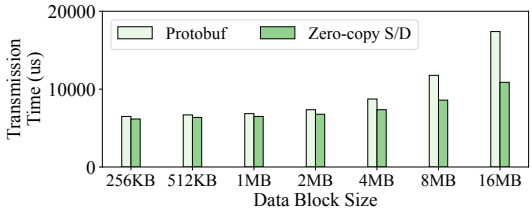


Fig. 8: Transmission time when sending a single data block with different sizes.

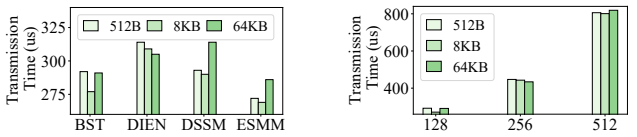
we should put data into the RDMA buffer as little as possible. In the context of this paper, we just put the tensors transferred to the remote GPU into the RDMA buffer, while leaving the tensors used by local CPU computation outside the RDMA buffer. Specifically, inference framework will use its original allocator when it is generating the tensors of the memory-intensive part of the model, while using the unified memory allocator provided by Zero when generating the intermediate tensors which will be transferred to the GPU pool.

Zero-copy Serialization and Deserialization. Although the tensors can be directly transferred with RDMA, there are still redundant data copies during serialization and deserialization because the original design of the message object includes the tensors. Taking Figure 7(a) as an example: when generating the message object from a tensor (e.g., a two-dimensional array in the figure), the message object includes the metadata of this tensor (dimension and element type of this tensor) and the tensor itself, which leads the first data copy. Then the message object is serialized into the binary data stream: the serialization library traverses every field of the message object and encodes the field. Generally, a field will be encoded into two parts—the description (e.g., the data type of this field) and the content of this field. Consequently, the tensor is copied once again. Similarly, the tensor will be copied one more time when the tensor is extracted from the binary data stream during the deserialization on the GPU node.

To eliminate those copies, we define a new data type (*rawstring*) to replace the tensor in the message object, as shown in Figure 7(b). The *rawstring* contains the pointer of the tensor (memory address of the tensor). Although the *rawstring* will be copied during the generation, serialization and deserialization of the message object, the cost is negligible. This is because the size of a pointer is just eight bytes and will not increase as the size of the tensor gets larger.

We evaluate the effectiveness of the above zero-copy serialization and deserialization: we transfer a single data block with different sizes, during which using Protobuf and the above proposed method (denoted as "Zero-copy S/D") to perform serialization and deserialization, and respectively record the overall transmission time. The results are shown in Figure 8. The proposed method outperforms Protobuf in all cases and brings more benefits as the data block size gets larger.

Besides, because the tensor and metadata need to be respectively processed on the GPU and CPU, we transfer the



(a) Different Models (b) Different BST Batch Sizes

Fig. 9: Transmission time with different merging thresholds.

serialized message object containing the *rawstring* to the main memory of the destination GPU node, and transfer the tensor to the GPU global memory of the destination node.

C. Dynamic Tensor Merging Policy

This section describes how we merge tensors to improve transmission efficiency. We first demonstrate why a dynamic merging threshold is required and then introduce how to locate the appropriate threshold.

Necessity of Merging Tensors Dynamically. As discussed in Section III-C, merging small tensors into a large data block may improve transmission efficiency. Because merging data blocks requires copying data blocks, which brings extra cost, merging tensors does not always offer advantages. Some prior studies [23], [39], [40] noticed that issue and only merged the data block whose size is not larger than a threshold (e.g., using 512B as the threshold [23]). However, a fixed threshold cannot maximize the transmission efficiency. We find that the appropriate threshold is not identical in different models: as shown in Figure 9(a), when running BST/DIEN with batch size 128 and running DSSM/ESMM with batch size 512, the highest transmission efficiency is achieved with the threshold set to 8KB, 64KB, 8KB and 8KB, respectively. On the other hand, the batch size also has impacts on the merging threshold: as shown in Figure 9(b), the appropriate threshold is 8KB, 64KB, 8KB for BST when the batch size is set to 128, 256 and 512, respectively. According to our statistics, the distribution of intermediate tensor sizes varies across different models or different batch sizes; this is why the appropriate threshold changes as the workload changes.

Locating Appropriate Threshold. We develop a policy to appropriately set the threshold; it can be done offline and does not incur the runtime overhead in Zero.

Given that there are many tensors $(t_0, t_1, \cdots, t_n)$ to be transferred with RDMA while the size of t_i is s_i, the transmission time of t_i is $Tr(s_i)$, and the time of copying t_i is $Cp(s_i)$, we can calculate the profit $P(T_m)$ when the merging threshold is set to T_m:

$$P(T_m) = Tr(\sum_{s_i \le T_m} s_i) + \sum_{s_i \le T_m} Cp(s_i) - \sum_{s_i \le T_m} Tr(s_i) \quad (1)$$

The first item and second item in Equation 1 calculate the cost of merging all tensors t_i whose sizes are smaller or equal to the threshold T_m into a large data block and transferring the merged data block. The third item in the equation calculates the cost of separately transferring every tensor t_i. With this equation, we can calculate the profits of a series of thresholds, and choose the threshold which brings the maximum profit. For example, given all intermediate tensor sizes $(s_1, s_2, \cdots, s_n)$ of a query, we can calculate the profit when using each s_i as the threshold.

The information required to calculate the profit can be obtained by offline profiling. Under given batch size of a recommendation model, the distribution of the intermediate tensor sizes is constant. We can therefore obtain the distribution by submitting a query to the model. The transmission/copying time can be obtained by profiling the transmission/copying time of man-made data blocks with different sizes.

After merging some tensors into a new data block, the new block instead of the original tensors will be transferred. We modify the *rawstring* of the tensor that is merged, or this tensor cannot be correctly located when deserializing the binary data stream on the GPU node. Specifically, the *rawstring* is modified to the memory address of the new data block plus the tensor's offset in the data block.

One may note that it is possible to eliminate the cost of merging data blocks: by pre-allocating some slots with given sizes in the RDMA buffer and letting the inference framework store tensors in the slots, the tensors are naturally organized into large data blocks. Since such an approach requires relatively heavy modification to the inference framework, we decide not to adopt it in Zero.

D. Building Effective GPU Pools

Based on the design of Zero and our investigation from production, we provide some implications on how to build a GPU pool with high performance and low total cost of ownership (TCO). To build a GPU pool, we should determine the appropriate number of GPUs on each GPU node, choose the proper networks between CPU nodes and GPU pool, and choose proper RDMA primitives if RDMA network is used.

The key guide of choosing specifications of each GPU node is making GPUs achieve peak utilization, and saturate the the network interface card (NIC) bandwidth at the same time. Note that the GPU utilization and the size of data that is required to be transferred are affected by the hosted application (the model), we can profile the common applications that will frequently run on the GPU pool. Given N models (denoted by $1, 2, \cdots, n$), the size of data to be transferred per second (denoted by S_i), the GPU utilization (denoted by U_i), the available bandwidth of the NIC (denoted by BW_i), the appropriate number of GPUs on a GPU node (denoted by NG_i) can be roughly calculated by $NG_i = (U_i/S_i) \times BW_i$.

Sometimes we need to use GPU nodes with fixed GPU cards already existing in the cluster to build the GPU pool. In this case, the goal is to minimize the number of GPUs used to build the pool while hosting all models, and this can be achieved by solving an optimization problem.

Given M GPU nodes (denoted by $1, 2, \cdots, m$) and the number of GPUs on node i is C_i, we introduce two variables: y_i and x_{ij}. If node i is used to build the pool, then y_i is 1, otherwise it is 0. If model j is deployed on node i, then x_{ij}

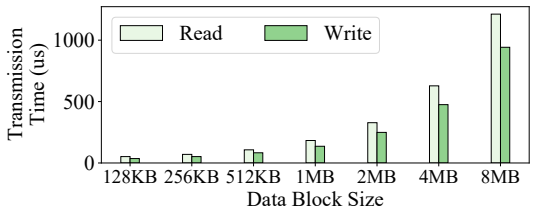


Fig. 10: Transmission time of data blocks with different sizes.

is 1, otherwise it is 0. The optimization problem is described with Equation 2:

$$\begin{aligned} Objective = \quad & MIN(\sum_{i=1}^{m} y_i \times C_i) \\ \text{Constraint-1:} \quad & \sum_{j=1}^{n} x_{ij} \times U_j \leq y_i \times C_i, \qquad i = 1, \cdots, m \\ \text{Constraint-2:} \quad & \sum_{j=1}^{n} x_{ij} \times S_j \leq BW_i, \qquad i = 1, \cdots, m \\ \text{Constraint-3:} \quad & \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \cdots, n \end{aligned} \tag{2}$$

Model configuration systems (e.g., Morphling [28] and Falcon [29]) can be used to collect the needed data above.

Choosing CPU and GPU Pool Networks. RDMA requires expensive hardware support, maintenance and development costs is high. Instead, using the TCP network does not require configuring new hardware, which can reduce acquisition costs. According to our measurements, with the support of high-performance network components like DPDK [55], if the total amount of transferred data is smaller than 1MB, the end-to-end latency of a recommendation query is similar with RDMA or TCP networks. In this case, TCP network is a better option for connecting the CPU nodes with the GPU pool.

Choosing RDMA Primitives. When using RDMA to transfer data, there are two available one-sided primitives (Read and Write). In the RPC transmission with RDMA, the RPC client and RPC server establish bilateral communication in the beginning. The client notifies the server of how much data in total will be transferred, and the server then allocates memory with the corresponding size. After that, the data transferring process starts. With Read primitive, the server is responsible for transferring data (read data from the client); with Write primitive, the client is responsible for transferring data. Since transferring data consumes CPU resources, the main difference between Read primitive and Write primitive is that one of the client/server bears the CPU resource pressure.

In general, the Write primitive is a better option for two reasons. Firstly, Read primitive is not a good choice when the CPU resource of the GPU node is insufficient (Section III-A) and the GPU node is the RPC server. Secondly, Read primitive performs worse than Write primitive according to our experiments. Figure 10 shows the time when we transfer different sizes of a data block with Read primitive and Write primitive, respectively. The Write operations are issued signaled, and Read and Write operations are combined with a combination of interrupt-based notification and polling. As we can observe, Write primitive outperforms Read primitive in all cases. The results are consistent with the research in X-RDMA [56].

V. Implementation

While we implement Zero based on TensorFlow (inference framework), Protobuf [17] (serialization library), and our in-house RPC framework (similar to eRPC [25]), Zero does not rely on any specific characteristics of them.

In TensorFlow, to guide the inference framework to call the appropriate memory allocator when generating tensors, we introduce a new attribute (a key-value pair) in the model description file (a .pbtxt file). After slicing the graph of a model into a CPU subgraph and a GPU subgraph, the attribute is automatically added to each output node of the CPU subgraph. When TensorFlow generates the output tensors of a node, it checks whether the node has the attribute or not; if it does, the memory allocator provided by Zero is used to allocate memory for the output tensors, or the default memory allocator of TensorFlow is used. We also add two member functions in *class Tensor* to enable type conversion between the tensor and *rawstring*; these functions will be used in Protobuf. The design principle of Zero can also benefit other inference frameworks that manage their own memory space independently when using the GPU pool. For instance, PyTorch maintains the memory space by itself [54].

In Protobuf, because we introduce a new data type *rawstring*, we need to ensure that this new data type works correctly and efficiently. Specifically, we enhance Protocol Buffer Compiler *protoc* [57] to enable assigning value to *rawstring*. To make the new data type *rawstring* compatible with the built-in features of Protobuf, we supplement the support of *rawstring* in the features—*RepeatedField*, *Reflection*, *dynamic_message* and *extensions*—of Protobuf. During the serialization process, Protobuf traverses each data of the message object and calls the corresponding serialization function of each data type. The same process occurs during the deserialization process. We supplement the serialization function *WriteRawString()* as well as the deserialization function *FillRawString()* for *rawstring*.

Moreover, to ensure that the serialized message objects can be transferred with RDMA directly, we utilize the *ZeroCopyStream* mechanism [58]. This mechanism provides an interface *Next()* which offers a custom memory allocation method for Protobuf. We customize the *Next()* interface to provide RDMA-registered memory and empirically allocate 10KB memory in each invocation. In *WriteRawString()*, the unused parts of the allocated RDMA-registered memory will be freed to avoid memory waste.

In RPC framework, there is a list indicating which data blocks will be transferred. Before transferring, we check the sizes of data blocks from the list to merge some blocks according to the merging threshold to improve transmission efficiency. To avoid introducing redundant data copies on destination GPU nodes, we send tensors to the GPU global memory of the destination using GPUDirect RDMA [59] technique, and store the destination address into the corresponding *rawstring*; the destination GPU node can extract this address by deserializing the serialized message object.

TABLE I: Experimental specifications

	Specification
Hardware	CPU: Intel Xeon Platinum 8369B @ 2.90GHz GPU: Nvidia A100-SXM4-80GB NIC: Mellanox ConnextX-5 adapters 2x100Gb/s
Software	OS: CentOS 7 with kernel 5.10 CUDA: Driver 470.154, SDK 11.4 Library: TensorFlow 1.15, Protobuf 3.8

VI. Evaluation

In this section, we evaluate the effectiveness of Zero with different recommendation models. We also introduce the deployment effect of Zero in production.

A. Experimental Setup

Table I summarizes the hardware and software configurations in the experiments. We evaluate Zero with a CPU server and a GPU server. Every CPU/GPU node is equipped with two Intel Xeon Platinum 8369B processors and communicates via Mellanox ConnextX-5 NICs. The MIG is enabled to create multiple GPU instances to simulate multiple GPU nodes.

In the experiments, four recommendation models (BST [5], DIEN [4], DSSM [6] and ESMM [7]) are used. When using a model for inference, we send large amounts of inference queries at a fixed rate and measure the end-to-end latency. We slice each model into the embedding layer and other layers; we run the embedding layer on the local CPU node and the remaining layers on the remote GPU nodes.

We compare Zero with the default Protobuf [17] and Cornflakes [23]. With Protobuf, the intermediate tensors will be transferred directly to the GPU pool, while Cornflakes merges some intermediate tensors in the transmission. We set the merging threshold as its proposed value (512B) when using Cornflakes. The queries are routed to GPU nodes using the JSQ policy [21] in the experiments.

B. Overall Evaluation

We first evaluate Zero in the scene where a single CPU node hosting a recommendation model sends inference queries to three GPU instances (4g.40gb, 2g.20gb and 1g.10gb). In this experiment, we use batch sizes that do not incur the SLA violation (40ms, according to our production requirements).

Figure 11 shows the results in different models with different batch sizes. As we can see, Zero outperforms Protobuf and Cornflakes in all cases. When respectively running BST, DIEN, DSSM and ESMM, Zero—compared with Protobuf—can shorten the average latency by 23.1% (up to 32.0%), 16.7% (up to 23.3%), 18.3% (up to 29.3%) and 16.0% (up to 28.0%). Compared with Cornflakes, Zero can shorten the average latency by 10.5% (up to 12.0%), 11.4% (up to 15.1%), 10.1% (up to 12.6%) and 8.3% (up to 10.6%).

The reason why Zero outperforms Protobuf and Cornflakes in terms of end-to-end latency is that Zero decreases the communication overhead in the RPC communication, as shown in Figure 12. Firstly, while Zero eliminates all the data copies during the RPC communication, the other two techniques bring

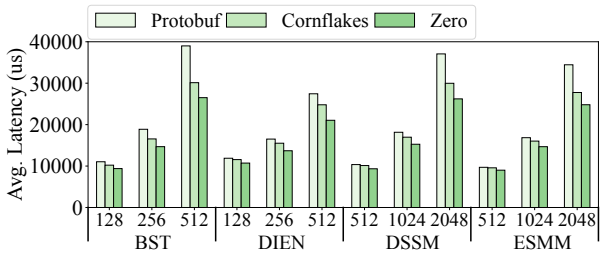


Fig. 11: Avg. latency with different models and batch sizes.

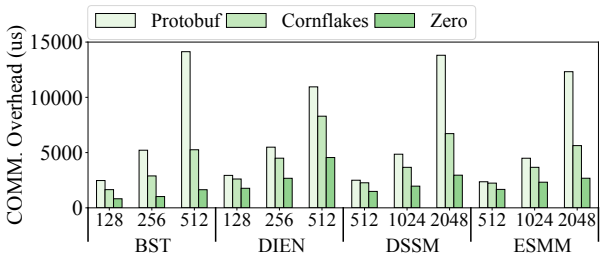


Fig. 12: Communication overhead corresponding to Figure 11.

data copies. With Protobuf, the intermediate tensors need to be copied four times during the RPC communication, as shown in Figure 1. With Cornflakes, the intermediate tensors need to be copied twice. The first data copy is introduced on the CPU node because Cornflakes requires to copy intermediate tensors from the inference framework to the RDMA buffer. The second data copy is introduced on the GPU node because using Cornflakes to send a serialized message cannot adaptively distribute the tensor and metadata to the GPU global memory and main memory of the destination GPU node. Secondly, Zero appropriately set the merging threshold for different workloads. Since Protobuf does not merge tensors, and Cornflakes merges tensors with a fixed threshold (as shown in Figrue 9, it cannot bring ideal efficiency), they bring lower transmission efficiency than Zero.

Besides, we can observe that Zero shows a larger advantage when the batch size gets larger. For example, when the batch size is set to 128, 256 and 512 for DIEN, compared with Protobuf, Zero reduces the average latency by 9.8%, 17.1% and 23.3%, respectively; compared with Cornflakes, Zero reduces the average latency by 7.3%, 11.7% and 15.1%, respectively. This is because when the batch size gets larger, the amount of transferred data is increased: in DIEN, the amount of transferred data is 2.2MB, 4.3MB, and 8.6MB under the batch size 128, 256, and 512. In this case, the communication overhead increases in Protobuf and Cornflakes because they need to copy more data.

In conclusion, Zero brings lower end-to-end latency compared with Protobuf and Cornflakes due to reducing the communication overhead more effectively in GPU pools.

C. Ablation Study

We conduct experiments to individually evaluate the effectiveness of the zero-copy transmission mechanism and tensor

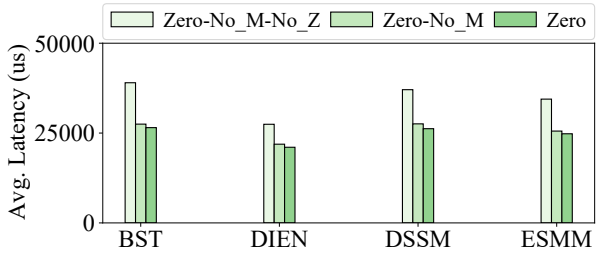


Fig. 13: Avg. latency in different models.

merging policy. The experimental setup remains the same with Section VI-B, and the batch sizes of the four models (BST, DIEN, DSSM and ESMM) are respectively fixed to 512, 512, 2048 and 2048. We implement two variations of Zero: 1) Disabling the two components of Zero (denoted as "Zero-No_M-No_Z"), and 2) disabling tensor merging policy in Zero (denoted as "Zero-No_M"). Figure 13 shows the average latency in different models.

Since the difference between Zero-No_M-No_Z and Zero-No_M is whether the zero-copy transmission mechanism is enabled or not, the effectiveness of the zero-copy transmission mechanism can be evaluated by comparing Zero-No_M-No_Z with Zero-No_M. As shown in the figure, the proposed zero-copy transmission mechanism can significantly reduce the latency in all cases. Specifically, the latency can be reduced by 29.5%, 20.2%, 25.7% and 25.8% in BST, DIEN, DSSM and ESMM, respectively.

By comparing Zero-No_M with Zero, the effectiveness of the tensor merging policy is evaluated. According to our proposed tensor merging policy, the merging threshold in this experiment is respectively set to 8KB, 64KB, 8KB and 8KB for BST, DIEN, DSSM and ESMM for high transmission efficiency. As we can see in the figure, the tensor merging policy reduces the latency by 2.5%, 3.1%, 3.6% and 2.1% in BST, DIEN, DSSM and ESMM. The benefits brought by the tensor merging policy in these four models are not significant, as the number of intermediate tensors in these models is not large (only a few dozen), resulting in fewer opportunities to merge small tensors. In our production model, there can be hundreds of intermediate tensors, so the policy yields greater benefits (detailed later in the next subsection).

In general, the zero-copy transmission can bring more benefits than merging small tensors. This is because the zero-copy transmission saves the multiple copying time of the large tensors, while merging small tensors saves the transmission time of the small tensors. The copying time of large tensors is longer than the transmission time of the small tensors.

D. Impact of GPU Node Configurations

We additionally study the impact of GPU node configurations. We use three dedicated A100 cards instead of three A100 MIG instances (as in Section VI-B) as the GPU nodes. While the A100 cards are connected via NVLink [60] in our experiment, the recommendation models we currently use can be fully loaded onto a single GPU, meaning that an inference query is only processed on one GPU. In this case, data is only transferred between CPUs and GPUs rather than across GPUs. Since Zero is designed for optimizing the RPC transmission between CPUs and GPUs, the connections between GPUs do not affect the performance of Zero.

Under the new configuration, compared with Cornflakes, we observe that Zero can decrease the latency of BST query by 9.4%, 14.5%, and 15.7% under the batch size 128, 256, and 512. Note that under the old MIG-style GPU node configuration, Zero decreases the latency by 8.1%, 11.3%, and 12.0% under the corresponding batch sizes. It shows that the benefit of Zero increases under the new configuration. We observe similar results in other recommendation models. The reason is that the computing power of each GPU node has been improved under the new configuration, which leads to an increased proportion of communication overhead in the end-to-end latency.

Overall, the benefit of Zero increases with the computing power of the GPU node.

TABLE II: Configuration of our production clusters

Distribution of Servers

	CPU Server	A10 Server	Eight-card A100 Server
Cluster 1	21.4%	65.5%	13.1%
Cluster 2	16.4%	81.2%	2.4%

E. Production Deployment

In this subsection, we demonstrate the benefits of deploying recommendation models in the GPU pool and the effectiveness of Zero in production.

Cluster Configuration. Table II shows the configurations of our production clusters that have CPU-only servers, single-card A10 GPU servers, and eight-card A100 GPU servers. The two clusters have around 3000 servers each. The single-card A10 GPU servers that are cost-efficient for inference are used for host a large number of business class recommendation models. When the loads of recommendation models are high, the eight-card A100 servers are also used to serve the inference queries based on Zero as GPU pools. With powerful A100 servers, the serviceability of the recommendation models is improved while saving a considerable amount of cost of purchasing new GPU servers.

Deployment Effect. Figure 14 presents the achieved throughput under the required SLA target (40ms). We hide the recommendation model names for commercial reasons. The figure shows four cases, running the models on a local A100 GPU (denoted as "Local" in the figure), on a local A100 GPU with 7 MIGs (denoted as "Local-M"), in GPU pool (denoted as "Pool"), and in GPU pool with 7 MIGs (denoted as "Pool-M"). Using GPU pool brings higher throughput because more CPUs can be used to process the embedding layers. Comparing Pool-M with Local-M, the throughput is improved by 1.9x to 2.5x when hosting different models.

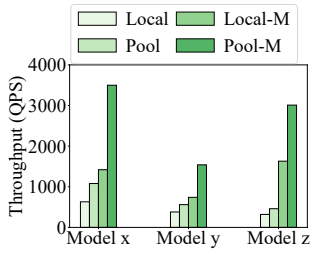


Fig. 14: Throughput of production recommendation models.

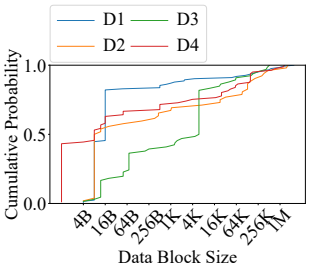
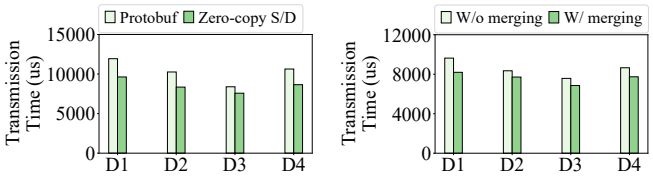


Fig. 15: The cumulative distribution diagram of the production data.



(a) Effectiveness of Zero-copy S/D (b) Effectiveness of W/ merging

Fig. 16: Transmission time of the production dataset.

We evaluate the effectiveness of the two components of Zero using actual production data. We collect intermediate tensors of our production models with different batch sizes deployed on the GPU pool to form four datasets (denoted as "D1" to "D4"). Figure 15 shows the distributions of the data of the four datasets. We first respectively transfer the four datasets, during which using Protobuf and our proposed zero-copy transmission mechanism ("Zero-copy S/D") to perform serialization and deserialization. The overall transmission time is reported in Figure 16(a). Our proposed technique decreases the transmission time by 19.3%, 18.6%, 9.7% and 18.6% in the four datasets. We then respectively transfer the four datasets without merging any tensor ("W/o merging"), and respectively transfer the four datasets while merging some tensors based on the proposed tensor merging policy ("W/ merging"). The transmission time is shown in Figure 16(b). For the four datasets, our proposed tensor merging policy can reduce the transmission time by 14.9%, 7.6%, 9.5% and 10.6%, respectively.

To summarise, GPU pooling enables cost-effectively enhancing the throughput of recommendation systems, and Zero also effectively reduces end-to-end latency in production.

VII. Conclusion

In this work, we propose a system named Zero to reduce the end-to-end latency of DNN-based recommendation systems deployed in GPU pools. It is proposed based on the findings that the long latency is mainly caused by the multiple data copies in RPC communication, and the inefficient transmission of multiple small tensors. Zero comprises a zero-copy transmission mechanism, and a dynamic tensor merging policy. The zero-copy transmission mechanism eliminates all the redundant data copies. The tensor merging policy appropriately merges small tensors into a large data block to improve transmission efficiency. Experimental results show that, compared with Protobuf and Cornflakes, Zero reduces the latency by 18.5% and 10.1% on average (up to 32.0% and 15.1%).

Acknowledgment

We sincerely thank all the anonymous reviewers for their valuable comments and constructive suggestions. This work is partially sponsored by the National Key Research and Development Program of China (2022YFB4501402) and the National Natural Science Foundation of China (62402316, 62232011, 62302302). Quan Chen and Guoyao Xu are the corresponding authors.

References

[1] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2020, pp. 488–501.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[3] M. Chui, J. Manyika, M. Miremadi, N. Henke, R. Chung, P. Nel, and S. Malhotra, "Notes from the ai frontier: Insights from hundreds of use cases," *McKinsey Global Institute*, vol. 2, 2018.

[4] 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 *Proceedings of the AAAI conference on artificial intelligence*, 2019, pp. 5941–5948.

[5] Q. Chen, H. Zhao, W. Li, P. Huang, and W. Ou, "Behavior sequence transformer for e-commerce recommendation in alibaba," in *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019, pp. 1–4.

[6] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 2333–2338.

[7] X. Ma, L. Zhao, G. Huang, Z. Wang, Z. Hu, X. Zhu, and K. Gai, "Entire space multi-task model: An effective approach for estimating post-click conversion rate," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2018, pp. 1137–1140.

[8] Q. Zheng, Q. Chen, K. Bai, H. Guo, Y. Gao, X. He, and M. Guo, "Bips: Hotness-aware bi-tier parameter synchronization for recommendation models," in *2021 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2021, pp. 609–618.

[9] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "Recshard: statistical feature-based memory optimization for industry-scale neural recommendation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 344–358.

[10] 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*. IEEE, 2022, pp. 1056–1070.

[11] L. Ke, X. Zhang, B. Lee, G. E. Suh, and H.-H. S. Lee, "Disaggrec: Architecting disaggregated systems for large-scale personalized recommendation," *arXiv preprint arXiv:2212.00939*, 2022.

[12] 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," *arXiv preprint arXiv:2001.02772*, 2020.

[13] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2021, pp. 802–814.

[14] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning." in *USENIX Symposium on Operating Systems Design and Implementation*, vol. 21, 2021, pp. 1–18.

[15] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning." in *USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 533–548.

[16] NVIDIA, "Nvidia a100 tensor core gpu architecture." https://www.nvidia.com/en-us/data-center/nvidia-ampere-gpu-architecture/.

[17] "Protobuf," https://protobuf.dev.

[18] J. Nelson and R. Palmieri, "Understanding rdma behavior in numa systems," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 2019, pp. 273–274.

[19] "Rdma over converged ethernet," https://www.ibm.com/docs/en/i/7.4?topic=concepts-rdma-over-converged-ethernet.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *2016 USENIX Annual Technical Conference*, 2016, pp. 437–450.

[21] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2p2: Making rpcs first-class datacenter citizens," in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019, pp. 863–879.

[22] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 462–478.

[23] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia *et al.*, "Cornflakes: Zero-copy serialization for microsecond-scale networking," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 200–215.

[24] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.

[25] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *16th USENIX Symposium on Networked Systems Design and Implementation*, 2019, pp. 1–16.

[26] M. Han, J. Hyun, S. Park, J. Park, and W. Baek, "Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2019, pp. 165–177.

[27] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1423–1431.

[28] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, "Morphling: Fast, near-optimal auto-configuration for cloud-native model serving," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 639–653.

[29] Y. Wu, H. Wu, D. Luo, Y. Xu, Y. Hu, W. Zhang, and H. Zhong, "Serving unseen deep learning models with near-optimal configurations: a fast adaptive search approach," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 461–476.

[30] 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.

[31] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir *et al.*, "Optimizing cpu performance for recommendation systems at-scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[32] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud gpus." in *USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 817–833.

[33] H. Fingler, Z. Zhu, E. Yoon, Z. Jia, E. Witchel, and C. J. Rossbach, "Dgsf: Disaggregated gpus for serverless functions," in *2022 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2022, pp. 739–750.

[34] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein, "Slashing the disaggregation tax in heterogeneous data centers with fractos," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 352–367.

[35] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over rdma," in *Proceedings of the Fourteenth EuroSys Conference*, 2019, pp. 1–14.

[36] "Flatbuffers," https://flatbuffers.dev.

[37] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free rdma networking in java," in *2021 USENIX Annual Technical Conference*, 2021, pp. 1–14.

[38] D. P. Proos and N. Carlsson, "Performance comparison of messaging protocols and serialization formats for digital twins in iov," in *2020 IFIP networking conference*. IEEE, 2020, pp. 10–18.

[39] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 195–211.

[40] T. Stamler, D. Hwang, A. Raybuck, W. Zhang, and S. Peter, "zio: Accelerating io-intensive applications with transparent zero-copy io," in *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 431–445.

[41] "Google cloud platform," https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/google-cloud-platform/.

[42] "Meta supercomputer," https://www.tomshardware.com/news/meta-supercomputer-16000-a100-gpus.

[43] "Azure gpu cluster," https://azure.microsoft.com/en-us/blog/azure-announces-general-availability-of-scaleup-scaleout-nvidia-a100-gpu-instances-claims-title-of-fastest-public-cloud-super/.

[44] "Tensorflow batching," https://github.com/tensorflow/serving/tree/master/tensorflow_serving/batching.

[45] "Aws pricing," https://aws.amazon.com/ec2/pricing/on-demand/.

[46] "Multi-instance gpu," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[47] "grpc," https://www.grpc.io/.

[48] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, "Racksched: A microsecond-scale scheduler for rack-scale computers." in *USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 1225–1240.

[49] R. R. Weber, "On the optimal assignment of customers to parallel servers," *Journal of Applied Probability*, vol. 15, no. 2, pp. 406–413, 1978.

[50] W. Winston, "Optimality of the shortest line discipline," *Journal of applied probability*, vol. 14, no. 1, pp. 181–189, 1977.

[51] G. Foschini and J. Salz, "A basic dynamic routing problem and diffusion," *IEEE Transactions on Communications*, vol. 26, no. 3, pp. 320–327, 1978.

[52] "Unified virtual memory," https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.

[53] "Bfcallocator," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/bfc_allocator.h.

[54] "Defaultcpuallocator," https://github.com/pytorch/pytorch/blob/main/c10/core/CPUAllocator.cpp.

[55] I. Cerrato, M. Annarumma, and F. Risso, "Supporting fine-grained network functions through intel dpdk," in *2014 Third European Workshop on Software Defined Networks*. IEEE, 2014, pp. 1–6.

[56] T. Ma, T. Ma, Z. Song, J. Li, H. Chang, K. Chen, H. Jiang, and Y. Wu, "X-rdma: Effective rdma middleware in large-scale production environments," in *2019 IEEE International Conference on Cluster Computing*. IEEE, 2019, pp. 1–12.

[57] "Protocol buffer compiler," https://github.com/protocolbuffers/protobuf?tab=readme-ov-file#protobuf-compiler-installation.

[58] "Protobuf zerocopystream," https://protobuf.dev/reference/cpp/api-docs/google.protobuf.io.zero_copy_stream/.

[59] "Gpudirect rdma," https://docs.nvidia.com/cuda/gpudirect-rdma/.

[60] "Nvidia nvlink," https://www.nvidia.com/en-us/design-visualization/nvlink-bridges.