Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services

Weihao Cui
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
weihao@sjtu.edu.cn

Mengze Wei
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
mzweilz@sjtu.edu.cn

Quan Chen
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Xiaoxin Tang
Shanghai University of Finance and Economics
Shanghai, China
tang.xiaoxin@sufe.edu.cn

Jingwen Leng
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
leng-jw@cs.sjtu.edu.cn

Li Li
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
lilijp@sjtu.edu.cn

Mingyi Guo
Dept. of CSE
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

***Abstract*—GPUs have been widely adopted to serve online deep learning-based services that have stringent QoS requirements. However, emerging deep learning serving systems often result in long latency, and low throughput of the inference request that damage user experience and increase the number of GPUs required to host an online service. Our investigation shows that the poor batching operation and the lacking of data transfer-computation overlap are the root causes of the long latency and low throughput. To this end, we propose Ebird, a deep learning serving system that is comprised of a GPU-resident memory pool, a multi-granularity inference engine, and an elastic batch scheduler. The memory pool eliminates the unnecessary waiting of the batching operation and enables data transfer-computation overlap. The inference engine enables concurrent execution of different batches, improving the GPUs resource utilization. The batch scheduler organizes inference requests elastically. Our experimental results on an Nvidia Titan RTX GPU show that Ebird reduces the response latency of inferences by up to 70.9% and improves the throughput by up to 49.3% while guaranteeing the QoS target compared with TensorFlow Serving.**

***Index Terms*—GPUs, DL Serving, Latency, Throughput, Responsiveness**

I. Introduction

Deep learning is famous for the high prediction accuracy and has been adopted in many online services that require short response time (e.g., intelligent personal assistant [1], online translation [2], and interactive photo editor [3]). GPUs have been proved to be particularly suitable for these computational demanding deep learning-based services, especially after the introduction of tensor cores in Nvidia Volta GV100 GPU architecture for speeding up neural network processing. It has been reported that GPUs can speed up the model training by more than 50× CPU [4]. Due to the high computational ability of GPUs, more and more service providers start to use GPUs to host the deep learning-based services [5]–[7].

For deep learning-based services, multiple inference requests are often organized and executed in batches, because a single inference cannot fully utilize all the resources of a GPU (e.g., the latest Nvidia Titan RTX has 72 SMs). Emerging deep learning serving systems, such as TensorFlow Serving [8], adopt a CPU-side batching mechanism to improve the inference processing throughput.

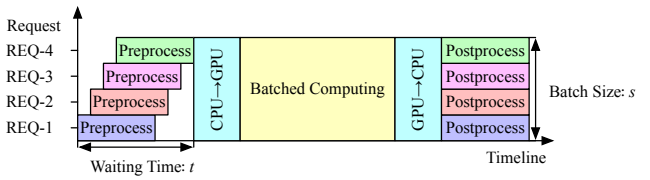

Fig. 1: Execution timeline of batched inference requests.

Generally, in most deep learning serving systems adopting CPU-side batching, inference requests are batched to gain high parallelism, as shown in Figure 1. Batch operations of input data are performed on the CPU side [8] [9] since the input of a deep learning network running on GPUs must be stored in a continuous address space. Then service providers can configure the maximum batch size s and the maximum waiting time t of an inference. Either the number of queued inferences reaches s, or the earliest inference waits for t, the queued inferences are organized to be a batch. When the last inference of a batch arrives, the input data of all the inferences are transferred to the GPU together. The GPU then processes the batched inferences together in a tight-couple way. After the processing completes, the results of all the inferences are transferred to the CPU together. Moreover, only after a batch of inferences returns, the next batch can be launched. This mechanism works well if the load of the deep-learning-based service is stable, and s and t are tuned carefully before the service starts based on the inference load.

However, online services often experience diurnal load pattern. Emerging batching mechanism results in long latency at low load and low throughput at high load. At low load, the response latency of the first inference in a batch is delayed

Quan Chen and Minyi Guo are corresponding authors.

by at least t (the processing time also increases due to the batching). At high load, due to the sequential processing of different batches, GPUs are idle when copying the result of inferences from GPU to CPU, and copying the input data of the inferences from CPU to GPU. The GPUs are not fully utilized even if the requests queued up seriously at the CPU side, resulting in the low throughput.

Eliminating unnecessary waiting at low load, and overlapping data transfer and computation at high load can improve the responsiveness and throughput of deep-learning-based services. However, if a short maximum waiting time t is adopted for eliminating the unnecessary waiting, each batch will have only a small number of inferences. When the load of the service bursts, the new inferences suffer from long latency. This is mainly because these new inferences are not launched to the GPU before the previous batch returns, even though the GPU is not fully utilized by the small batch of interferences. Configuring a short maximum waiting time of inference is not helpful in reducing the latency of inferences in online services (discussed in Section III).

The concurrent kernel execution feature [10] of the current GPUs that allows independent kernels in different CUDA streams[1] to run concurrently on different SMs of a GPU can be leveraged to solve the above problem. We observe that processing multiple inferences in a single large batch using a single CUDA stream has similar performance with processing these inferences using multiple streams with smaller batches. Therefore, if we can elastically launch multiple small batches of inferences to the GPU when the load bursts, the GPU can be better utilized even if the short maximum waiting time is adopted. The elastic batching also enables data transfer-computation overlap, thus improving the throughput.

Based on this observation, we propose **Ebird**, a novel deep learning serving system to improve the responsiveness and throughput of online deep learning-based services. Ebird is comprised of a *GPU-resident memory pool*, a *multi-granularity inference engine* and an *elastic batch scheduler*. The memory pool holds the input data of all the inferences. Whenever an inference is submitted, its input data (and other meta information) is directly transferred into the memory pool. The memory pool enables data transfer-computation overlap by transferring data in the backend when the GPU is processing other inferences. The multi-granularity inference engine provides multiple CUDA streams that process inference batches of different granularities, thus enabling concurrent kernel execution. The batch scheduler organizes the inferences in the memory pool into batches of different granularities elastically and schedules them to the appropriate workers in the engine. In Ebird, an inference is processed as soon as possible if there are available GPU resources to reduce the response latency. Our main contributions are as follows.

- **Comprehensive analysis of batch scheduling for deep learning-based services on GPU.** The analysis demonstrates that emerging batching policies result in long latencies and low throughput of online services.
- **A GPU-side inference batching mechanism.** We implement a novel GPU-side memory pool that stores the inputs of all the inferences in the GPU global memory. It enables transfer-computation overlap and elastic batching.
- **A novel elastic batch scheduling policy.** We design a multi-granularity inference engine and a corresponding batch scheduler that minimizes the response latency of inferences while improving the throughput of services.

Our experimental results on an Nvidia Titan RTX GPU show that Ebird reduces the response latency of inferences by up to 70.9% and improves the throughput by up to 49.3% while guaranteeing the QoS target compared with TensorFlow Serving (hereinafter called the "TF-Serving") running with optimized scheduling setup.

[1]A CUDA stream is a sequence of operations that execute in issued-order, while operations issued to different CUDA streams execute in parallel.

II. Related Works

Researchers have made efforts to develop GPU-based deep learning systems for particular purposes like better performance [11]–[14] or QoS management [15], [16]. Some works focus on the optimization in mainstream deep learning systems, including Tensorflow [17], Caffe [18], Pytorch [19], and others. Generally, offering services in datacenters only needs the forward computation of the whole deep learning model training process, which is implemented but not optimized for serving in the frameworks mentioned above. Although the accuracy of the model evaluating, and performance of training are two keys to deep learning research. The quality of service(QoS) and utilization of the full serving system play essential roles in providing deep learning services.

Systems for deep learning serving are developed according to the difference of properties between training and serving. Those systems such as Tensorflow Serving [8] and Clipper [9], integrate the mainstream model training frameworks to provide the inference engine. Thus they inherit performance problems caused by training. Such systems manage QoS by protecting the end-to-end tail latency with the batch size and waiting time for a batch at high-level, which is a coarse-grained way. Some works focus on specific deep learning models. BatchMaker [20], DeepCPU [21], and GRNN [22] are specially designed to improve inference speed for RNNs. These works are short of generalization and are not able to figure out the existing problems for all deep learning models.

Systems [15], [16], [23], [24] focusing on QoS management are more generalized for QoS task running on GPUs, where the end-to-end latency is controlled through API provided by Nvidia. However, QoS management at the API fails to take the deep learning serving properties into account. These works take into account the co-location of user-facing applications and batch applications on GPUs, which can be future work for us.

III. Background and motivation

In this section, we investigate the problems of existing deep learning serving systems for online services with diurnal load

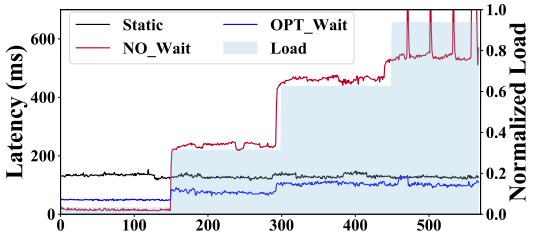

Fig. 2: The end-to-end latencies of the inferences with different batching policies when the load of the service bursts.

pattern. Without loss of generality, we use TF-Serving as the representative serving system and use Resnet_152 (Res152 in short) that is widely used in image classification services as the representative network to perform the investigation. To emulate the pattern, we increase the submit frequency of the inference requests for every 150 inferences.

A. Existing Problems

Figure 2 shows the end-to-end latencies of the inferences when different batching policies are adopted in TF-Serving. In the figure, the shadowed area shows the load variation of Res152, the x-axis shows the arrival order of the inferences, and the y-axis shows the latencies of the inferences. "NO_Wait" and "OPT_Wait" represent the policies that set the maximum waiting time of an inference request to 0 and 30ms, respectively. The optimal maximum waiting time is identified according to the official guide of TF-Serving [25]. For all the policies, the maximum batch size is 32, that is the recommended batch size for Res152 in many research papers [9]. "Static" policy is similar to "OPT_Wait", except the batch size is fixed to 32. If there are less than 32 valid inferences in a batch, the batch is padded to have 32 inferences with dummy inferences to better utilize the tensor cores in GPU.

Observed from Figure 2, NO_Wait achieves the shortest latency when the load is low but suffers from long latency at the high load that results in the Quality-of-Service (QoS) violation. On the contrary, OPT_Wait achieves much shorter latency at high load but suffers from relative long latency at low load. Meanwhile, the static policy always performs worse than the OPT_Wait policy. TF-Serving recommends the service providers to adopt the OPT_Wait batching policy.

To better understand how the batched inferences are processed on a GPU, Figure 3 presents the trace of processing inferences with the OPT_Wait policy at high load. The execution trace is captured with the official profiling tool *nvprof* [26] provided by Nvidia. In the figure, "HtoD" and "DtoH" represent the operations of copying data from main memory to GPU and from GPU to main memory, respectively. "Computation" represents the execution of the kernels.

Observed from Figure 3, the GPU is idle between adjacent batches. This is mainly because TF-Serving schedules different batches sequentially. Only after the result of the current batch is transferred to the main memory, the input data of the next batch can be transferred to GPU. The scheduling overhead

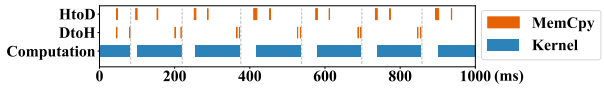

Fig. 3: Snapshot of inference processing with OPT_Wait. Inference processing with NO_Wait is similar except the kernels are shorter.

and the data transfer together result in the large idle gap. This figure also explains the reason that the NO_Wait policy results in long latency at high load. If NO_Wait is adopted, a batch often has a small number of inferences and cannot fully utilize the GPU. In this case, the inference requests queued at the CPU side will not be launched until the previous batch completes even if the GPU is not fully utilized. The resulted long queueing time is the root cause of the long latency at high load with the NO_Wait policy.

According to the above investigation, emerging deep learning serving system results in the long latency of inferences and the low processing throughput. The root causes of the two problems are the long waiting time for batching, the low GPU utilization due to the sequential processing of different batches, and the lacking of transfer-computation overlap.

B. The Ways to Solve the Existing Problems

A deep learning serving system that maximizes the throughput while satisfying the QoS target and minimizes the latency of inferences at low load is required to cater to the diurnal load pattern. We propose Ebird, an adaptive deep learning serving system to achieve the above purpose. According to the above analysis, Ebird should have the following abilities.

- **Ebird should be able to overlap data transfer and computation to minimize the GPU idle time between adjacent batches.** By keeping the SMs of a GPU busy, more inferences can be processed at high load. However, state-of-the-art systems have no input pipeline that can deliver data for the next batch when the current batch is being processed. Ebird needs to design a software mechanism to overlap the transfer and computation.
- **Ebird should be able to run multiple batches of inferences concurrently.** With this ability, when the load of the service bursts, the new inferences can be executed immediately if the GPU is not fully utilized.
- **Ebird should be able to organize inferences into batches of different granularities elastically.** This ability minimizes the waiting time of inferences and improves GPU utilization. When a GPU is processing a large batch of inferences, a small batch of inferences can be launched to utilize the remaining GPU resources and vice versa. State-of-the-art systems (e.g., TF-Serving) fix the maximum batch size during the lifetime of service.

IV. Methodology

In this section, we elaborate on the design overview of Ebird.

Figure 4 shows the design overview of Ebird, a deep learning serving system that is composed of a *GPU resident*

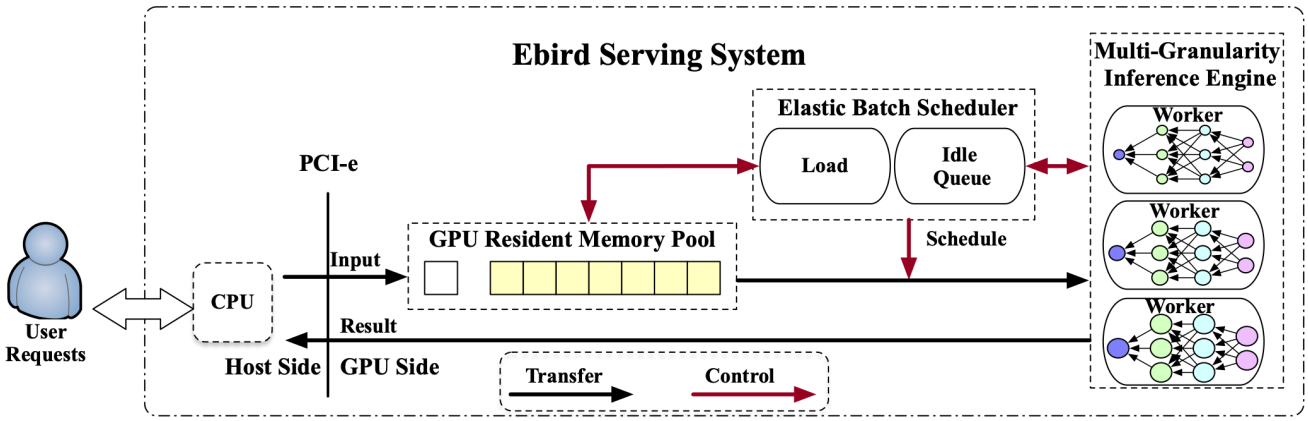

Fig. 4: Overview of Ebird serving system.

memory pool, a *multi-granularity inference engine*, and an *elastic batch scheduler*. The GPU resident memory pool keeps inputs of inference requests one by one in sequence. It enables data transfer-computation overlap. The multi-granularity inference engine maintains multiple workers that run and return the result to the host independently and concurrently. The workers are configured for inferences of different batch sizes. The elastic batch scheduler organizes the inferences in the memory pool into batches of different sizes elastically and assigns them to a suitable worker, based on the workload and the running states of the workers.

In more detail, when an inference request *inf* is submitted to Ebird, it is served in the following steps.

1) Once Ebird receives *inf*, it is immediately offloaded to the GPU by transferring the required information (e.g., input data, address of the return result, synchronization flags) to the GPU resident memory pool (Section V). The inference requests are sorted in the order of their arrival time. The challenge here is how to manage the inference requests from multiple users efficiently.
2) The elastic batch scheduler organizes the inferences in the memory pool into batches based on the running states of the workers in the multi-granularity inference engine (Section VII). When multiple workers for inference batches of different sizes are free, it is challenging to decide the way to batch the inferences so that their response latencies can be minimized.
3) If *inf* is organized into an inference batch of size n, it is scheduled to the free worker w for the inference batch of size n. After the worker w completes the batch of inferences, based on the return addresses of these inferences stored in the memory pool, the inference results are transferred to the main memory and returned to users (Section VI).

V. GPU Resident Memory Pool

In this section, we first discuss the design of the memory pool in detail. Finally, we validate the reasonability of designing the memory pool.

TABLE I: Parameters of slot in memory pool.

Parameters	Explanation
index	Serial ID of inference request
InDevPtr	Device address of input
InEvent	Input CUDA event
OutCpuPtr	Host address of output
OutEvent	Output CUDA event

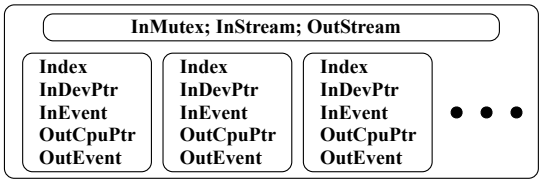

Fig. 5: Structure of GPU resident memory pool.

A. Design of GPU Resident Memory Pool

Considering the disadvantages of traditional batching operations, we design a GPU resident memory pool to replace the original batching operations on CPU.

The GPU resident memory pool acts as a circular buffer, which holds input data of different inferences in sequence in a continuous address of GPUs global memory. The memory pool keeps allowing transferring individual request input from CPU to GPU, instead of waiting until the last request in a batch comes. The memory pool transfers the input data of different requests serially in order of arrival. In this way, we can get the mapping from input to output, and return inference result to the corresponding request.

Figure 5 shows the structure of the memory pool. **InMutex** guarantees that only one inference's input is transferred at a time. **InStream** and **OutStream** are two CUDA streams that are responsible for communication between the memory pool and the multi-granularity inference engine. Requests are also responded through these two streams. Data in the memory pool are organized in slots. Each slot mainly contains five components, as listed in Table I. In order to transfer input data for an inference, **InStream** calls $cudaMemcpyAsync$ and records the corresponding **InEvent**. Suppose that a worker in the multi-granularity inference engine needs to process

4 slots input data (**index** from 0 to 3). Since **InEvent**s of **Slot 0 − 2** happen before **InEvent** of **Slot3**, this worker only monitors the occurrence of **InEvent** of **Slot3** to check whether all the four input data are already. When the worker finishes the computation task, results are output to the corresponding **OutCpuPtr**, and **OutEvent** is recorded into **OutStream**. As **OutEvent** occurs, the memory pool responds to the user. For efficient batching, the number of slots in the memory pool is recommended to be several times of **MaxInf**, which is the maximum number of alive inference requests as described in Table II.

B. Benefits of GPU Resident Memory Pool

Thanks to the GPU resident memory pool, the waiting time is excluded on the host, as the input data is transferred as soon as the request arrives. Requests get concatenated one by one automatically when entering the memory pool. No extra CPU resources are needed to keep the batch queue. The batch size is later determined by elastic batch scheduler.

The memory pool also brings benefits in terms of data transfer-computation overlap. The GPU resident memory pool acts as a buffer zone between the incoming requests and the scheduler. After being received, a request waits for its turn to get processed on GPU instead of queuing on the CPU. The worker in the inference engine directly fetches the ready input stored in the memory pool instead of waiting for data transfer.

C. Validating Reasonability

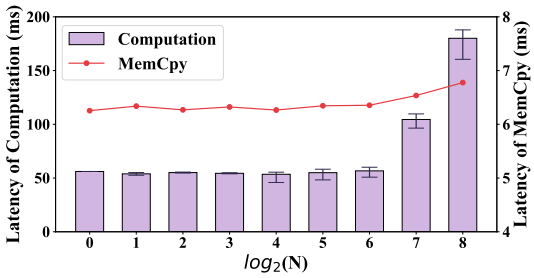

Fig. 6: Latency of split Memcpy and computation.

Despite theoretical benefits, there may be a doubt in the effectiveness of the GPU resident memory pool. Typically, transferring a single large file between disk and memory is faster than transferring multiple small files with the same total volume. Similarly, it is also possible that individual input data transfer through PCI-e declines performance.

To validate the reasonability of the memory pool, we conduct a simple experiment, in which 256 pictures are copied from CPU to GPU through PCI-e to simulate data transferring. A total of 256 pictures are divided into N fragments, where N may equal to $1, 2, 4, 8, 16, ...256$. The recorded elapsed time of transferring 256 pictures with different N is shown in Figure 6. The x-axis represents the binary logarithm of the number N. The right y-axis represents the latency of memory operations for transferring 256 pictures.

As we can see, splitting data movement into small batches has similar performance to data movement in a large batch. The latency of transferring a large piece of data and multiple pieces of data with the same total size through PCI-e are almost equivalent. The maximum difference of latency between with and without data splitting is lower than one millisecond (3.9 microseconds for each request), which is negligible.

Overall, our design philosophy of the memory pool is supported by this experiment.

VI. Multi-Granularity Inference Engine

In this section, we exploit the multi-granularity inference engine to enable multiple batches of inferences to run concurrently. We also validate the performance of the inference engine and discuss the configuration of multiple workers.

A. Enabling Concurrent Multiple Batches

The multi-granularity inference engine is aimed at adapting to bursty load. Multiple workers are kept alive simultaneously in the inference engine. Each worker can be configured with different batch sizes and run independently since they are bound to different CUDA streams. Therefore, the inference engine is capable of launching multiple workers to process the inferences according to the load. The batch size summation of all the busy workers increases in real time when the load raises.

The idle workers reside in a priority queue called idle queue, which is regularly updated by the scheduler introduced in Section VII. To cooperate with the scheduling policy, the workers in the idle queue are sorted in descending order according to its batch size by using a red-black tree. Each worker is responsible for processing the batched requests and returning the result of the inference to the host side. After finishing processing, the worker enters the idle queue.

B. Performance Validation

There is also a doubt in the performance of the multi-granularity inference engine. For instance, provided that the latency of running two workers of batch size 4 concurrently is much longer than one worker of batch size 8, there is a great possibility that the inference engine leads to QoS violation when running multiple workers to support a high load.

We conduct another simple experiment to validate the inference engine performance. In the experiment, the CUDNN convolution function, which is the most compute-intensive function in deep learning networks, is called repeatedly for 50 times to simulate inference of a deep learning network, what we call *FakeNet*. Assuming that 256 inferences of *FakeNet* are remaining to be processed, we complete all the inferences with N workers of batch size M, where $N * M = 256$ and N varies according to the list$(1, 2, 4, 8, ...256)$. The elapsed time of each possibility is shown in Figure 6. The x-axis represents the binary logarithm of the number N, while the left y-axis represents the latency of the inferences of *FakeNet*.

As shown in Figure 6, with the same amount of inferences, the computation latency of using one worker with single large batch size and using multiple workers with multiple small batch sizes are almost the same as long as we manage the

workers carefully. The computation latency maintains stable between 1 and $2^6 = 64$, while increases when the computation is divided into $2^7 = 128$ and $2^8 = 256$. This is because Titan RTX has 72 SMs, indicating that at most 72 CUDA warps can be executed at the same time. When more than 72 streams are in use concurrently, latency may increase owing to low utilization of each SM and serialization of different streams.

The experiment results show that the performance of the inference engine can get guaranteed as long as its configuration is carefully managed.

C. Configuration of Multiple Workers

Considering the experiment results above and the demand of the elastic batch scheduling policy, the batch size of alive models in the inference engine all coincide to 2^n, where n is a non-negative integer. Currently, given the maximum allowed batch size $s = 32$, we keep six models alive in the inference engine, whose batch sizes are configured as the list $(1, 1, 2, 4, 8, 16)$. This is based on the overall consideration of three factors. First, each integer can be produced by the list $(1, 1, 2, 4, 8.....)$. Thus the inference engine is capable to accommodate the different load. Second, with this configuration, a worker of large batch size can be scheduled to better utilize the parallelism of GPU under high load instead of using too many workers with small batch size. Third, If the batch size of all workers s are set to 1 to accommodate the different load, then the GPU global memory will be used up, even on the Titan RTX which has a large global memory of 24GB.

VII. Elastic Batch Scheduler

In this section, we introduce the elastic batch scheduler, which improves responsiveness and throughput by coordinating the memory pool and inference engine.

Algorithm 1 lists how the scheduler schedules the inference requests in the memory pool to be processed by the workers in the inference engine. The parameters used in the algorithm are listed in Table II. More specifically, *alive* in the table means that the inferences are in the process of computation.

TABLE II: Parameters of elastic batch scheduling algorithm.

Parameters	Explanation
DevPtr	Device address where input data begins
N	Number of ready input in memory pool
Q	Queue of idle workers
MaxInf	Maximum inferences allowed alive
CurInf	Number of alive inferences

The scheduler runs as follows. Firstly, when monitoring the memory pool and inference engine, the scheduler accesses the information about load ($\mathbf{N}$), the beginning device address ($\mathbf{DevPtr}$) of input remaining to be scheduled, and the number of alive inferences ($\mathbf{CurInf}$). Secondly, the scheduler works out that if there are idle workers and the maximum number of inferences that can be dispatched to the inference engine by choosing the smaller one ($\mathbf{R}$) of $\mathbf{N}$ and ($\mathbf{MaxInf} - \mathbf{CurInf}$). Then the scheduler repeatedly picks the first worker in the idle queue $\mathbf{Q}$ whose batch size is not greater than $\mathbf{R}$ until $\mathbf{R}$ is less than 0 or no workers can be picked. The scheduler switches the input address of the chosen worker to $\mathbf{DevPtr}$ and wakes up the worker from the idle queue $\mathbf{Q}$. The scheduler also updates the idle queue $\mathbf{Q}$ when workers finish processing inferences dispatched to them. The time complexity of the scheduling algorithm is $O(logM)$, where M is the number of the alive workers in the multi-granularity inference engine.

Algorithm 1 Elastic batch scheduling algorithm.

Require: $\mathbf{N}$, $\mathbf{DevPtr}$, $\mathbf{Q}$, $\mathbf{MaxInf}$, $\mathbf{CurInf}$

```
while True do
    if !Q.empty() then
        R ← min(N, MaxInf − CurInf)
        Woker ← Q.front()
        while R > 0 and Worker do
            if R ≥ Worker.batchsize then
                Schedule DevPtr → Worker.input
                Worker.run()
                R ← R − Worker.batchszie
                DevPtr ← DevPtr + Worker.batchsize
                Q.remove(Worker)
            else
                Worker ← Worker.next()
```

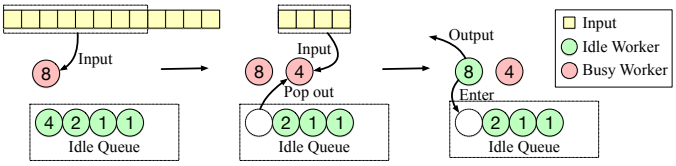

Fig. 7: Example of elastic batch scheduling.

Figure 7 shows an example of how the elastic batch scheduler coordinates the memory pool and the inference engine work. ▭ represent concatenated input data in the memory pool. ○ represent busy worker which are performing inference, while ○ represent idle worker. Assume that at a certain time, input data of 12 inferences are ready in the memory pool. A worker with batch size 8 has been scheduled to process the first 8 inferences, while 4 requests remain in the memory pool. The batch size of the first worker in the idle queue is 4. At the next scheduling, the scheduler pops the first worker out from the idle queue and schedules this worker to process the remaining 4 requests. Later, when the worker of batch size 8 completes the inference, the scheduler puts the worker back into the idle queue.

Through such work style, the scheduler operates with the information from memory pool and the inference engine. The batch size configuration varies in real-time according to the load and the GPUs operation status. There is a balance between the memory pool and the inference engine.

VIII. Evaluation

In this section, we evaluate the effectiveness of Ebird in improving the responsiveness and the throughput while satisfying the QoS requirement of deep learning-based services.

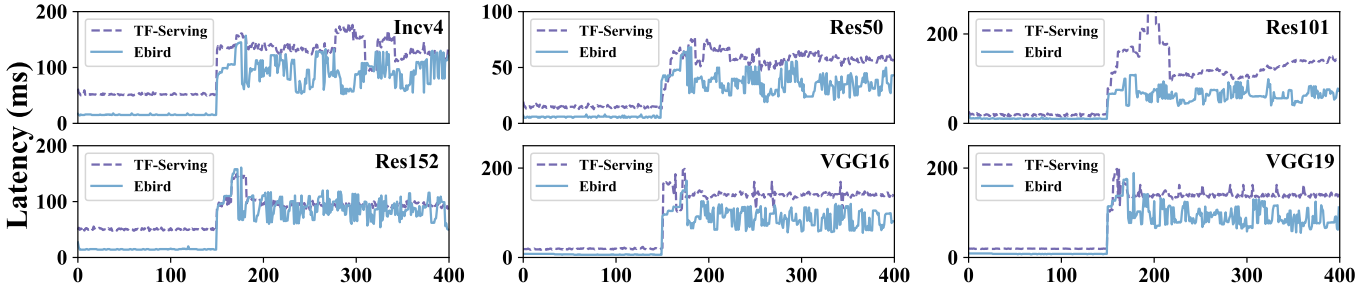

Fig. 8: The end-to-end latencies of the inferences in different benchmarks with TF-Serving and Ebird.

A. Experiment Setup

We perform all the experiments on a machine equipped with the latest Nvidia Titan RTX GPU. The GPU has 72 SMs and 576 Tensor cores, and has been shown to be able to deliver outstanding performance for deep learning inferences [27]. Table III lists the detailed experimental setup.

As shown in Table III, we use six widely-used deep neural networks as the online services to evaluate Ebird. Note that, Ebird does not rely on any specific features of these benchmarks, thus is suitable for other deep learning-based services. We compare Ebird with state-of-the-art deep learning serving system, TF-Serving, in the following of this section. TF-Serving uses the OPT_Wait policy described in Section III for all the benchmarks because it has been shown to be able to provide better performance than NO_Wait policy and the static policy. That is, the maximum batch size is set to 32, and the maximum waiting time is set to be an optimized value for each benchmark.

TABLE III: Evaluation specifications.

CPU	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
GPU	NVIDIA TITAN RTX (72 SMs, 576 Tensor Cores)
OS	Ubuntu 16.04.5 LTS with kernel 4.15.0-51-generic
Software	GPU Driver Version: 418.39 CUDA Version: 10.1; CUDNN Version: 7.5
Benchmarks	Inceptionv4 (Incv4); Resnet_50 (Res50); Resnet_101 (Res101) Resnet_152 (Res152); VGG_16 (VGG16); VGG_19 (VGG19)

B. Improving Responsiveness

In this experiment, we evaluate the effectiveness of Ebird in improving the responsiveness of deep learning-based services with diurnal load pattern. To emulate the diurnal load pattern, we launch 400 inference requests for every benchmark, in which the first 150 inferences are launched in the low rate, and the later 250 inferences are launched in the high rate. The load is high if the latencies of the inferences are close to the QoS target (200*ms* is used in this experiment) with TF-serving.

Figure 8 shows the end-to-end latencies of the inferences in different benchmarks when the inferences are served with TF-Serving and Ebird, respectively. Observed from this figure, Ebird can significantly reduce the end-to-end latency of the inferences at both low load and high load for all the benchmarks compared with TF-Serving. When the load is low, Ebird reduces the latency of the inferences ranging from 44.6% to 70.9% for the benchmarks. When the load is high, Ebird reduces the latency of the inferences ranging from 7.4% to 53.1% for the benchmarks.

The reason why Ebird can reduce the latency of the inferences at low load is that it reduces the unnecessary waiting time. Besides, Ebird is able to improve the responsiveness at high load because it processes inferences using multiple independent workers in the multi-granularity inference engine. An inference can be processed once there are free workers, and once a worker completes its inferences, the inference results are immediately returned to the users. On the contrary, even though the waiting time of inferences is short at high load with TF-Serving, the inference results are returned after all the inferences in the current batch complete. Because the processing time of a large batch of inferences is long, early inferences in a batch suffer from longer response latency with TF-Serving compared with Ebird.

Moreover, TF-Serving results in QoS violation of the inferences in Res101, when the load increases. This is mainly because TF-Serving processes batches of inferences sequentially. When the current load is low, the inferences are organized into small batches. If the load increases dramatically, the inferences queue up even if the currently running batch is not able to fully utilize the GPU. The queuing results in the long end-to-end latency of the inferences when the load bursts. On the contrary, Ebird is able to process the bursty inferences if the current inferences are not able to fully utilize the GPU. Ebird can always guarantee the QoS of deep learning-based services no matter the load is bursty or not.

C. Increasing Throughput while Guaranteeing the QoS

In this subsection, we evaluate Ebird in increasing the throughput of inference processing while guaranteeing the QoS. We use stable load in this experiment to eliminate the impact of load bursty on the latencies of the inferences.

Figure 9 presents the achieved inference processing throughput with TF-Serving and Ebird while the latencies of the inferences are shorter than the QoS target. As we can see from this figure, Ebird improves the inference throughput of all the benchmarks compared with TF-Serving. On average, Ebird improves the throughput by 34.4% compared with TF-Serving. In this way, given the same peak load of a deep learning-based service, fewer GPUs are needed to host the service with Ebird.

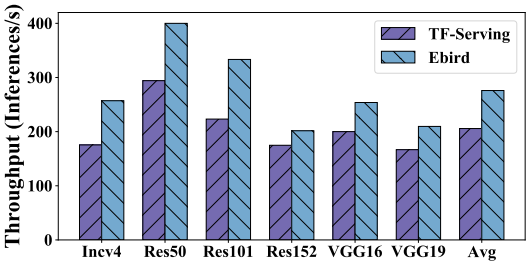

Fig. 9: The inference processing throughput of the benchmarks with TF-Serving and Ebird while guaranteeing the QoS.

Ebird is able to improve the throughput while guaranteeing the QoS because it overlaps data transfer and computation. On the contrary, the SMs in the GPU are idle when the input data/the inference result is transferred to/from the GPU. Observed from Figure 9, the throughput improvements are high for some benchmarks (e.g., Res101) but are relatively low for other benchmarks (e.g., Res152). This is mainly because the benchmarks have different data transfer-computation ratios. The data transfer-computation ratio of Res101 is higher than the corresponding ratio of Res152. If the time of data transfer takes a large percentage of an inference's end-to-end latency, overlapping data transfer and computation eliminates the long GPU idle time due to data transfer.

D. Diving into Ebird

To better understand why Ebird performs better than TF-Serving, Figure 10 shows the execution trace of executing inferences of Res152 with Ebird (Figure 3 shows a similar trace with TF-Serving). In Figure 10, "Worker-*n*" shows the kernel execution in the worker for inference batches of size n, "Whole GPU" shows all the kernel execution in all the workers on the whole GPU.

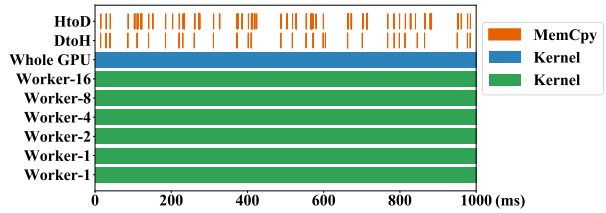

Fig. 10: Snapshot of inference processing with Ebird.

Comparing Figure 10 and Figure 3, the inputs of inferences are transferred to GPU separately in Ebird, while TF-Serving transfers the input data of all the inferences in a batch together. The separate data transfer is enabled by the GPU resident memory pool that stores inputs of all the inferences. In this way, the data transfers are distributed on the execution timeline and do not interrupt the computation of GPU. Because data transfer and computation overlap with each other, the GPU is always processing kernels at high load, as shown in Figure 10 (Row "Whole GPU").

Observed from Figure 10, we can also find that the six workers run in parallel, while the kernel execution timeline of each worker is relatively sparser than that in Figure 3. If the kernel from one worker can occupy all SMs of the GPU, the kernels from other workers are not executed until there are idle SMs on the GPU. The kernel execution timeline of worker with the smaller batch size is also sparser than that of worker with larger batch size, indicating that the idle worker queue intends to schedule a worker with larger batch size under high load. The data transfers from the GPU to the main memory are also scattered on the timeline, which are executed by each worker. It explains why Ebird is able to reduce the end-to-end latency of inferences at high load, as shown in Figure 8.

E. Overhead of Ebird

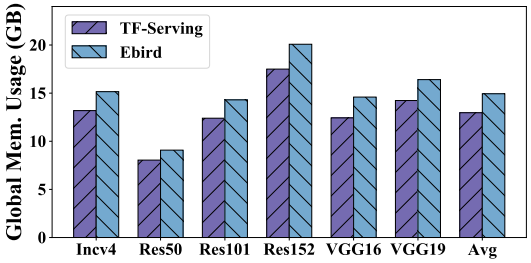

Fig. 11: Global memory usage of TF-Serving and Ebird.

The overhead of Ebird comes from the multi-granularity inference engine owing to maintaining multiple inference workers. Figure 11 shows the global memory usage of TF-serving and Ebird. As we can see, Ebird uses 15.2% more global memory space compared with TF-Serving. Ebird uses more global memory space because workers duplicate the global memory used for storing the weight of deep learning network in our current implementation. Moreover, the extra global memory [28] needed by convolution is also duplicated. It can be relieved by sharing weight between workers because there is no need to update the weight when performing inferences.

IX. Conclusion

Ebird improves responsiveness and throughput for deploying deep learning services in datacenters outfitted with GPUs. For these purposes, Ebird enables the GPU-side prefetch mechanism and the elastic batch scheduling policy for the deep learning serving system. Through comparing the performance of Ebird and TF-Serving (State-of-the-art deep learning serving system), we verify the effectiveness of Ebird in eliminating the waiting time for responsiveness and overlapping data transfer and computation for GPUs when providing deep learning services. Generally, Ebird enhances responsiveness. Moreover, Ebird improves the throughput by 34.4% on average compared with state-of-the-art solutions, TF-Serving. In the future, we will implement the weight sharing mechanism to reduce global memory usage overhead.

Acknowledgment

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004800), the National Natural Science Foundation of China (NSFC) (61602301, 61632017, 61832006, 61702328).

REFERENCES

[1] Apple siri. [Online]. Available: https://www.apple.com/siri/

[2] Google translate. [Online]. Available: https://translate.google.com/

[3] Prisma. [Online]. Available: https://prisma-ai.com/

[4] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[5] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[6] Big basin v2. [Online]. Available: https://code.fb.com/ml-applications/the-next-step-in-facebook-s-ai-hardware-infrastructure/

[7] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang, "Multi-tenant gpu clusters for deep learning workloads: Analysis and implications," Technical report, Microsoft Research, 2018. https://www. microsoft. com/en , Tech. Rep., 2018.

[8] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[9] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.

[10] N. Corporation, "Cuda c/c++ streams and concurrency." [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[11] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[12] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Acm Sigplan Notices*, vol. 52, no. 8. ACM, 2017, pp. 193–205.

[13] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines," *arXiv preprint arXiv:1512.06216*, 2015.

[14] Z. Fang, T. Yu, O. J. Mengshoel, and R. K. Gupta, "Qos-aware scheduling of heterogeneous servers for inference in deep neural networks," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 2067–2070.

[15] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[16] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 17–32, 2017.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, 2017.

[20] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency rnn inference with cellular batching," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 31.

[21] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 951–965.

[22] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "Grnn: Low-latency and scalable rnn inference on gpus," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, p. 41.

[23] Z. Wei, C. Weihao, K. Fu, Q. Chen, M. Daniel, Edward, W. Bo, L. Chao, and G. Minyi, "Laius: Towards latency awareness and improved utilization ofspatial multitasking accelerators in datacenters," in *Proceedings of the 33rd ACM international conference on Supercomputing*. ACM, 2019, pp. 58–68.

[24] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 483–496, 2017.

[25] "Tensorflow serving batching guide," 2019. [Online]. Available: https://github.com/tensorflow/serving/tree/master/tensorflow_serving/batching

[26] N. Corporation, "Profiler users guide." [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[27] "Nvidia turing architecture whitepaper," 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[28] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.