E^2bird: Enhanced Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services

Weihao Cui, Quan Chen, Han Zhao, Mengze Wei, Xiaoxin Tang, Minyi Guo

Abstract—We aim to tackle existing problems about deep learning serving on GPUs in the view of the system. GPUs have been widely adopted to serve online deep learning-based services that have stringent QoS(Quality-of-Service) requirements. However, emerging deep learning serving systems often result in poor responsiveness and low throughput of the inferences 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 lack of data transfer-computation overlap are the root causes of the poor responsiveness and low throughput. To this end, we propose E^2bird, 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 GPU resource utilization. The batch scheduler organizes inferences elastically to guarantee the QoS. Our experimental results on an Nvidia Titan RTX GPU show that E^2bird reduces the response latency of inferences by up to 82.4% and improves the throughput by up to 62.8% while guaranteeing the QoS target compared with TensorFlow Serving.

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

1 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 inferences 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). Table 1 depicts the details about the correlation between batch size and throughput by profiling the execution of different deep learning models with different batch sizes. In Table 1, BS represents the batch size of each model, Lat represents the latency for processing such a batch, and Req/s represents the corresponding throughput in the form of the amount of the inferences processed per second. Three models are profiled, including Resnet_50, Resnet_101, and Resnet_152(Res50, Res101, and Res152 in short). In all cases, the latency and throughput increase with the growth of the batch size used in processing inference batch. Owing to the positive correlation between latency and throughput, there

TABLE 1: Correlation of batch size, latency, and throughput.

Model	Res 50		Res101		Res152	
BS	Lat	Req/s	Lat	Req/s	Lat	Req/s
4	6.8	588	11.8	338	17.3	231
8	12.1	661	20.6	388	29.2	273
16	21.5	740.	36.2	441	49.5	323
32	40.1	798	65.4	489	93.7	341

exists a trade-off between latency and throughput. From a user's perspective, the priority is processing inference requests in a smaller batch to get quicker responsiveness. Nevertheless, the deep learning service providers prefer to processing larger batches to support the higher load with the same hardware resources. An efficient batching policy is needed to retain high throughput while guaranteeing the quality of deep learning services.

Emerging deep learning serving systems, such as TensorFlow Serving [8], adopt a CPU-side batching mechanism to improve the inference processing throughput. Generally, in most deep learning serving systems adopting CPU-side batching, inferences 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

• *All the authors are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.*

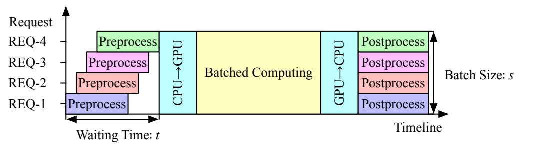


Fig. 1: Execution timeline of batched inference requests.

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 a diurnal load pattern. Emerging batching mechanism results in poor responsiveness at low load and low throughput at high load. At low load, the response latency of the first inference in a batch is delayed 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 input data of the inferences from CPU to GPU and copying the result of inferences from GPU to CPU. 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 inferences. Configuring a short maximum waiting time of inference is not helpful in reducing the latency of inferences in online services (discussed in Section 3).

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 when scheduled with reasonable strategies. Therefore, if we can elastically launch multiple inference batches of different batch sizes 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 **E^{2}bird**, a novel deep learning serving system to improve the responsiveness and throughput of online deep learning-based services. E^{2}bird 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. The batch scheduler can be configured with different scheduling policies.

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.
- **Novel elastic batch scheduling policies.** We design a multi-granularity inference engine, and a corresponding batch scheduler, which consists of two scheduling algorithms that minimizes the response latency of inferences while improving the throughput of services.

Our experimental results on an Nvidia Titan RTX GPU show that E^{2}bird reduces the response latency of inferences by up to 82.4% and improves the throughput by up to 62.8% while guaranteeing the QoS target compared with TensorFlow Serving (hereinafter called the "TF-Serving") running with optimized scheduling setup.

E^{2}bird is our follow-up work of Ebird, which has been published at the 37th IEEE International Conference on Computer Design (ICCD'19). It is different in the following aspects from our previous paper:

- **E^{2}bird attests to the theoretical benefits of the GPU resident memory pool.** Through this, E2bird can precisely predict the maximal improvement for different models, which provides a suggestion when optimizing the system.
- **E^{2}bird exploits several techniques to reduce the global memory overhead caused by multiple inference workers.** E^{2}bird is capable of configuring the alive workers flexibly in the inference engine by reusing memory for intermediate results and sharing the weight parameters between workers.
- **E^{2}bird abstracts a scheduling model for guiding the efficient scheduling.** E^{2}bird analyzes the interference between inference workers by the qualitative approach.
- **E^{2}bird provides a new scheduling policy.** The performance of different deep learning models varies

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

with the architecture of deep learning models and GPUs. The new proposed scheduling policy leverages an offline phase to get the best configuration of the elastic batch scheduler for different models instead of the coarse-grained scheduling in Ebird.

- **E^{2}bird extends the experimental evaluation.** E^{2}bird achieves an improvement of throughput by 9.9% averagely compared with Ebird.

2 Related Work

In this section, we discuss the state-of-the-art techniques and their limitations in three aspects.

2.1 Traditional QoS Management on GPUs

There has been a lot of work about traditional QoS management on GPUs. Baymax [11] is the first one that identifies the root causes for QoS violation on GPUs. Baymax [11], Prophet [12], and Flep [13] focus on using either a runtime system or a compilation engine to achieve QoS goals at a software level. With the emerging of MPS [14], Laius [15] targets eliminating the QoS violation on spatial multitasking accelerators such as Nvidia Volta GV100 GPU. To get a generalized solution, the management and scheduling units in the above previous work are all at the level of kernel functions of GPUs. Scheduling at the level of kernel functions brings in scheduling overhead for each kernel function, which is inevitable for deep learning services with too many kernel functions in it. These systems [11]–[13], [15], where the end-to-end latency is controlled through API provided by Nvidia fail to take the deep learning serving properties into account. However, they all take into account the co-location of user-facing applications and batch applications on GPUs, which can be future work for us. Fine-grained QoS [16] aims to propose QoS mechanisms for a fine-grained form of GPU sharing. Its key idea is that multiple kernels share the same SM to improve utilization. Due to no support of hard preemption and context reset on real hardware, its implementation is based on the simulator, GPGPU-Sim [17] instead of real hardware. Hence, the simulation property makes it unable to apply to current deep learning services.

2.2 Optimizations in Deep Learning Systems

Researchers have made an effort to develop GPU-based deep learning systems for particular purposes like better performance [18]–[22]. Some works focus on the optimization in mainstream deep learning systems, including Tensorflow [23], Caffe [24], Pytorch [25], 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 and utilization of the full serving system play essential roles in providing deep learning services.

Many projects [8], [9], [26], [27] provide the capability of deep learning serving. Clipper [9] is a modular architecture which builds on existing deep learning frameworks. Clipper introduces techniques including caching, batching, and adaptive model selection to reduce inference latency and improve throughput on CPUs and GPUs. To support frameworks such as Spark [28], Tensorflow [23], and so on, Clipper adopts to serve the deep learning model in the CPU-based container. Such coarse-grained management leads to low GPU utilization. Nexus [26] is another deep learning serving framework, which focuses on the accelerating on a GPU cluster. Nexus divides the origin whole deep learning model into fragments of deep learning models. Then, Nexus uses several batching techniques to guarantee the QoS target when deploying multiple sub-models to a GPU cluster. Nexus emphasizes the large service scale. Tensorflow Serving [8] is the model serving version of Tenserflow, which provides the mechanisms including load balance, model versioning, and QoS protection. Tensorflow Serving provides several batching guides [29] for users to guarantee the QoS when serving, which will be discussed detailedly in the rest of this paper. The batching policies employed in the above frameworks all lie in the CPU-side batching mechanism where E^{2}bird concentrates to give a better solution.

2.3 Accelerating Deep Learning Computation

Much work has focused on accelerating the inference of particular deep learning models. BatchMaker [30], DeepCPU [31], and GRNN [32] are all specially designed to improve inference speed for RNNs. Accelerating particular models depends on particular properties. Therefore, such work is short of generalization and is not able to figure out the existing problems for all deep learning models. Some work uses model compression techniques to reduce the size of the deep learning model and accelerate its inference. A couple of developing trends are pruning [33], [34], and low-bit quantization [35]–[37]. Optimization, like accelerating deep learning inference, can be regarded as complementary to E^{2}bird to get better performance.

3 Background and Motivation

In this section, a typical pipeline of deep learning serving is given out firstly. Then we use experiments to dig out the existing problems and expound the direction to solve them.

3.1 Typical Pipeline

A typical pipeline of deep learning service includes the following steps. The serving systems load models, process inference when receiving the inference requests, and return corresponding inference results. Simultaneously, the serving systems update the parameters of models in the background when older models can not guarantee enough accuracy.

Hence, deep learning serving covers a broad array of traditional online services and deep learning. Deep learning serving systems provide modules [8] that service RPC requests to carry out inference using loaded models. In such modules, QoS, load balance, resource isolation, and high utilization are fundamental issues, which is similar to traditional online services. Meanwhile, deep learning models must be updated basing on the real-time data. Therefore, deep learning serving systems have some new features, including model versioning, model loading. In the rest of

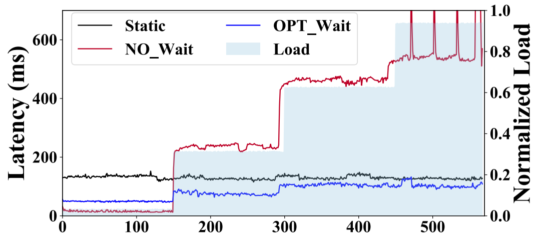


Fig. 2: The end-to-end latencies of the inferences with different batching policies when the load of the service bursts.

the paper, we focus on the trade-off between QoS and high throughput for accelerating deep learning serving systems on GPUs.

3.2 Existing Problems

We investigate the problems of existing deep learning serving systems for online services with a diurnal load pattern. Without loss of generality, we use TF-Serving as the representative serving system and use Resnet_152 (Res152 in short), 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.

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 [29]. For all the policies, the maximum batch size is 32, which 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.

As 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 QoS violation. On the contrary, OPT_Wait achieves much shorter latency at high load but suffers from relatively 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* [38] 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.

As shown in Figure 3, the GPU is idle between adjacent batches. This is mainly because TF-Serving schedules

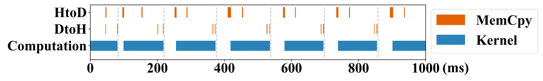


Fig. 3: Snapshot of inference processing with OPT_Wait. Inference processing with NO_Wait is similar except the kernels are shorter.

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 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, the emerging deep learning serving systems result 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.

3.3 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 E^{2}bird, an adaptive deep learning serving system to achieve the above purposes. According to the above analysis, E^{2}bird should have the following abilities.

- **E^{2}bird 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 a 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. E^{2}bird needs to design a software mechanism to overlap the transfer and computation.
- **E^{2}bird should be able to run multiple batches of inferences concurrently.** Sharing computing and memory is enabled. With this ability, when the load of the service bursts, the new inferences can be executed immediately if the GPU is not fully utilized.
- **E^{2}bird 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.

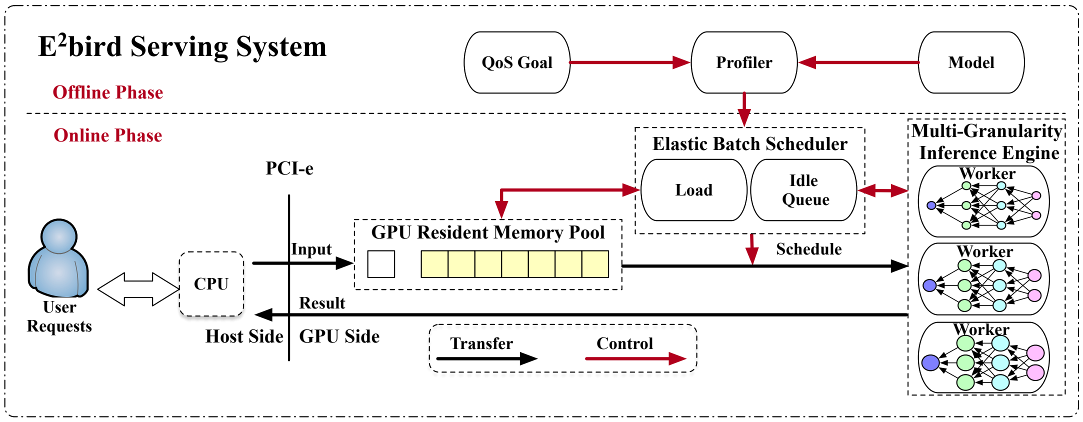


Fig. 4: Overview of E^{2}bird serving system.

4 Methodology

In this section, we elaborate on the design overview of E^{2}bird. Figure 4 shows the design overview of E^{2}bird, a deep learning serving system that is composed of an *offline profiler*, a *GPU resident memory pool*, a *multi-granularity inference engine*, and an *elastic batch scheduler*.

The workflow of the system is divided into two phases. **In the offline phase**, essential information for scheduling is profiled by executing the deep learning models under different constraints, including QoS goal, GPU platforms and so on. Then the elastic batch scheduler and multi-granularity inference engine are configured with the collected information.

In the online phase, 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 processing inferences in 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 load and the running states of the workers. The elastic batch scheduler works with two scheduling algorithms, named N-Ebird and E-Ebird. In the offline phase, N-Ebird only needs the maximum batch size allowed for guaranteeing the QoS target. However, E-Ebird needs extra offline profiling to get the best configurations of the inference engine under different load for each deep learning model.

5 GPU Resident Memory Pool

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

5.1 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 GPU's 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.

TABLE 2: 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

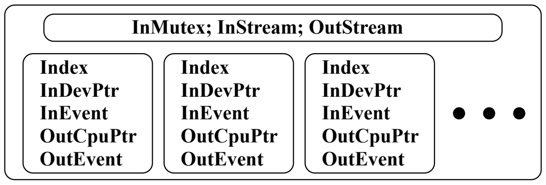


Fig. 5: Structure of GPU resident memory pool.

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 2. 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 four slots input data (**index** from 0 to 3). Since **InEvent**s of **Slot** $\mathbf{0-2}$ happen before **InEvent** of **Slot3**, this worker only monitors the occurrence of **InEvent** of **Slot3** to check whether all the four input data are ready. It is noted that all **InDevPtr**s in adjacent slots belong to a contiguous global memory space, which enables

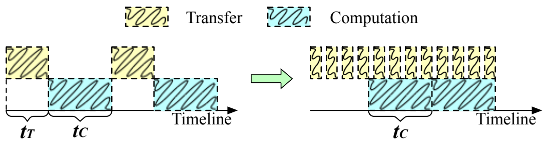


Fig. 6: Improvement of GPU resident memory pool.

the input data can be organized into batches of different sizes. 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 3.

5.2 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 the elastic batch scheduler.

The memory pool also brings benefits in terms of data transfer-computation overlap. As aforementioned, we get a new execution timeline of the batched requests. The left side of Figure 6 shows the origin round-robin way of execution. The right side of Figure 6 shows how the batched requests are executed theoretically after introducing the memory pool. The GPU resident memory pool acts as a buffer zone between the incoming requests and the scheduler. The received 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.

$$\lambda = \frac{t_T}{t_T + t_c} = \frac{1}{1 + \frac{t_c}{t_T}} \tag{1}$$

Comparing the two execution timelines, Equation 1 expresses the upper limit of the theoretical throughput improvement. Let λ, t_T, and t_C represent the ratio of the throughput increase, data transfer time, and computation time. Analyzing this equation gives us some hints: ❶ The efficiency of the memory pool dominates the defacto throughput improvement. With a lower overhead of memory pool, E^{2}bird can get closer to the upper limit; ❷ The ratio of computation time to data transfer time($\frac{t_c}{t_T}$) determines the upper limit. Thus the throughput improvement varies with the $\frac{t_c}{t_T}$ of different deep learning services. The deep learning services with low $\frac{t_c}{t_T}$ get greater improvement than those with high $\frac{t_c}{t_T}$.

5.3 Validating Reasonability

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.

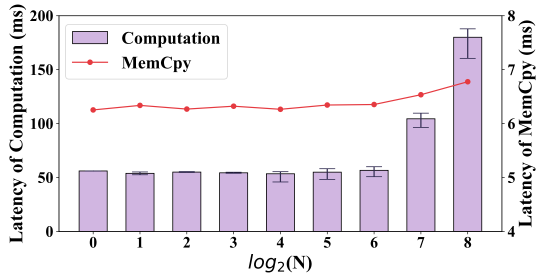


Fig. 7: Latency of split Memcpy and computation.

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 7. 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. The performance stability of the memory pool is guaranteed in spite of the brought-in overhead.

6 Multi-Granularity Inference Engine

In this section, we exploit the multi-granularity inference engine to enable multiple inference workers to run concurrently. We also adopt techniques to reduce the global memory overhead caused by multiple workers, validate the performance of the inference engine, and discuss the configuration of multiple workers.

6.1 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 rises.

The idle workers reside in a priority queue called idle queue, regularly updated by the scheduler introduced in Section 7. 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, which will be re-sorted automatically.

6.2 Reducing Global Memory Overhead

Without careful global memory management, keeping multiple workers alive in the inference engine consumes much global memory space of GPUs. In those deep learning training systems, several techniques have been developed to relieve global memory consumption, including dynamic memory allocation, re-computation, and memory swapping [39], which have negative impacts on performance. These techniques are all kinds of a trade-off between memory and performance, which is not applicable to deep learning serving. The passion for high performance is the first rule when it comes to online service. We mainly make good use of two *static* memory optimizations to reduce global memory consumption while guaranteeing the high performance of inference workers [40] . Afterward, the inference engine avoids the trade-off.

6.2.1 Reuse memory for intermediate results

Commonly, the deep learning model holds a large number of hidden layers in it. However, only a single layer is active for computation on GPUs at the same time. In deep learning serving, intermediate results of hidden layers do not get involved in any back-propagation in compared to training. There is no need to store the intermediate results that no succeeding layers depend on. These properties allow us to reuse the global memory for computation of the currently active layer, allocated for preceding or succeeding layers. When referring to reusing the global memory, we statically reuse the allocated global memory in multiple tensors instead of using a unified memory pool. Consequently, reusing the global memory in the workers relieves the overuse of global memory brought in by multiple workers without a negative impact on the performance of inference.

6.2.2 Weight sharing among workers

All the workers in the multi-granularity inference engine provide the same deep learning service based on the same model. The weight parameters of the deep learning model can be shared among all the workers. Owing to the read-only property, only a copy of weight parameters is enough for all workers, which further alleviates the global memory overuse.

The static memory allocation runs in the following steps. The multi-granularity inference engine holds all the weight parameters of the deep learning model, which can be accessed by all the workers in the engine. Each inference worker uses two APIs(*ScanNetwork(), AllocateGpu()*) to conduct the static memory allocating for intermediate results. *ScanNetwork()* scans the model architecture to get the computation topology, on which the worker can construct the computation dependencies based. Finally, *AllocateGpu()* allocates the global memory for each tensor in the worker before serving deep learning inference request.

6.3 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 7. 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 7, with the same amount of inferences, the computation latency of using one worker with a 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 mainly because that too many CUDA streams running together cause too much context switch overhead of SMs, which results in severe performance degradation.

The above experiment results show that the performance of the inference engine can get guaranteed as long as its configuration is carefully managed. Considering the experiment results above and the demand of the elastic batch scheduling policy in Section 7, the batch size of all workers in the inference engine all coincide to 2^n for better utilization of hardware, where n is a non-negative integer. Note that the configuration details of the multi-granularity inference engine will be discussed in Section 7, which varies according to the requirements of the elastic batch scheduler.

7 Elastic Batch Scheduler

In this section, we first use a scheduling model to qualitatively derive the rules that the elastic scheduling policy should obey to guarantee the QoS. Then we introduce the elastic batch scheduler, which improves responsiveness and throughput by coordinating the memory pool and inference engine. The elastic batch scheduler consists of two elastic scheduling policies.

7.1 Rules for Guaranteeing the QoS

The multi-granularity inference engine enables E^{2}bird to launch a new inference worker whenever there are sufficient inference requests and hardware resources. Certainly, the latency time of inference requests changes due to co-running of former and new inference workers. So the elastic scheduling policy must guarantee that all the busy inference workers meet the requirement of the QoS.

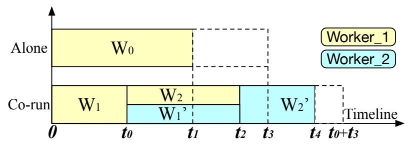


Fig. 8: Interaction of co-running batches.

For the above reason, the QoS guaranteeing of the E^{2}bird turns into a more complex problem of multi-QoS. For the qualitative analysis, we abstract a simplified scheduling model from the co-running of multiple workers in the multi-granularity inference engine. This simplified scheduling model only consists of two inference workers: the already-launched one and the new-launched one. The reason for that is, all the already-launched workers can be considered as a big worker who meets the requirement of the QoS without the interference of the new-launched worker.

Figure 8 shows the execution timeline for the simplified scheduling model. Here, the 🟨 represents *worker_1* which consists of all the already-launched workers and the 🟦 represents the new-launched *worker_2*. We set the QoS target to the time interval t_3. As shown in Figure 8, the **Alone** timeline means that without the interference of new-launched worker, *worker_1* completes the computation at t_0, which is definitely shorter than t_3. Within the **Co-run** timeline, *worker_1* runs independently from 0 to t_0. At t_0, *worker_2* is launched according to the elastic scheduling policy. From t_0 to t_1, *worker_1* and *worker_2* runs jointly on GPUs. At t_1, *worker_1* completes computations, and *worker_2* begins to run independently. Finally, *worker_2* completes computations at t_4. To guarantee the QoS, both the elapsed time of *worker_1* and *worker_2* should be shorter than the QoS target t_3, which is equivalent to Equation 2.

$$\begin{cases} t_2 < t_3 \\ t_4 < t_0 + t_3 \end{cases} \tag{2}$$

To simplify the resource sharing mechanism on GPUs, we firstly conduct an experiment to characterize the performance of deep learning models on GPUs. Here, we still use the convolution operator to simulate the deep learning models, which occupies the most of computation time. A convolution operator with batch size ranging from 1 to 64 is executed for 152 times in the experiment. The increase of batch size indicates the increase in serving workload. And two kinds of convolution algorithms are used -winograd [41] and im2col [42]- which are mostly used in the SOTA deep learning models. The latencies of the 64 cases are recorded in Figure 9. The x-axis represents the batch size used for executing, and the y-axis represents the corresponding latencies. As we can see, the latency of im2col grows linearly as the batch size increases. While the latency of winograd grows as a "ladder", the general growing trend is still linear. Hence, for the qualitative analysis, we can assume that the execution time of the worker can be obtained by $\frac{W}{u}$, where W means the workload and u means the compute capacity provided by GPUs.

During the execution, we divide the workload W of a worker into multiple fractions according to the change of compute capacity u. When multiple workers run jointly, the workers share the total compute capacity u of the GPU. Then we can get Equation 3. Here W_0 and W_0' represent the total workload of $worker_1$ and $worker_2$. Respectively, W_1, W_1', W_2, and W_2' represent the workload of the different periods, as shown in Figure 8. u_0 represents the total compute capacity of the GPU. u_1 and u_2 represent the compute capacity that $worker_1$ and $worker_2$ get during co-running. The summation of u_1 and u_2 should be smaller than u.

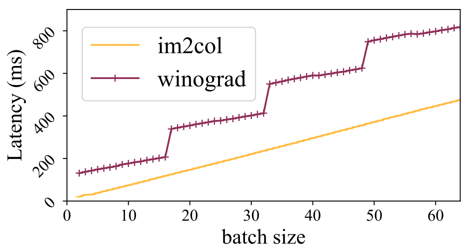


Fig. 9: Performance Scalability of Deep Learning Operators

$$\begin{cases} W_0 = W_1 + W_2 \\ W_0' = W_1' + W_2' \\ u_0 \geqslant u_1 + u_2 \end{cases} \tag{3}$$

$$\begin{cases} u_0 \geqslant u_1 + u_2 \\ \frac{W_1}{u_0} + \frac{W_2}{u_1} < t_3 \\ \frac{W_1'}{u_2} + \frac{W_2'}{u_0} < t_3 \end{cases} \tag{4}$$

Equation 4 can be obtained by combining Equation 2 and Equation 3. After some transformation, we conclude as Equation 5. Only if Equation 5 is satisfied, the QoS can be guaranteed. As we can see, u_2 is restricted by the upper limit and the lower limit to meet the QoS target of both workers. Or said differently, elastic scheduling policy aims to control the compute capacity of the new-launched worker to guarantee the QoS.

$$\begin{cases} u_2 < u_0 + \frac{W_2 u_0}{W_1 - t_3 u_0} \\ u_2 > \frac{W_1' u_0}{t_3 u_0 - W_2'} \end{cases} \tag{5}$$

7.2 Naive Elastic Scheduling: N-Ebird

On instinct, the batch size is the main factor that influences the compute capacity of an inference worker. A worker with a larger batch size tends to have higher parallelism, which means occupying more SMs when running jointly with other workers. Therefore, in the naive elastic scheduling policy, we control the total amount of current active inference requests to improve the responsiveness and get a higher throughput while guaranteeing the QoS, which is the original design of Ebird.

N-Ebird ensures that the total amount of active requests are smaller than a specific value at a coarse-grained level regardless of controlling the amount of the inference workers. This specific value is considered to be the max batch size (*MaxBS*) that can be used by an inference worker while not violating the QoS, which is profiled offline. We launch the new inference worker as long as the bath size summation of older workers and the new-launched worker is smaller than *MaxBS*. This simple thought is based on the performance

TABLE 3: 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
MemState	State of GPU resident memory pool (only used in E-Ebird)
EngineConfs	Different engine configurations for various load (only used in E-Ebird)
CurConf	Current configuration of multi-granularity inference engine (only used in E-Ebird)

validation of the multi-granularity inference engine in Section 6.3 and Equation 5. Firstly N-Ebird avoids an excessive total number of allowed workers in the inference engine in order not to degrade the performance when compared with using a single inference worker of large batch size. Also, the restriction on the number of active requests prevents the system from violating the QoS caused by GPU resource contention.

We discuss the detailed mechanism about N-Ebird in combination with the scheduling model in Section 7.1. In N-Ebird, we assume that each inference request shares the GPU resource equally. Meanwhile, if the QoS target is set at the latency when requests are processed in a batch of size *MaxBS*, the batch size summation of *worker_1* and *worker_2* in Figure 8 can not surpass *MaxBS* under the scheduling of N-Ebird. Basing on the hypothesis of equal resource sharing, *worker_1* and *worker_2* can get the compute capacity, which depends on batch size. So the performance of the concurrent *worker_1* and *worker_2* can be retained as that of a single worker whose batch size equals to *MaxBS*, which means Equation 5 is satisfied . Now that *worker_1* is launched ahead of *worker_2* in fact, the QoS can be guaranteed.

7.2.1 Configuring inference engine

The configuration of the multi-granularity inference engine under the scheduling of N-Ebird is as follows. 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 of accommodating the different loads. 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 sizes of all workers s are set to 1 to accommodate the different load, the GPU global memory is overused, let alone the poor performance of the inference engine.

7.2.2 Scheduling Algorithm

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 3. More specifically, *alive* in the table means that the inferences are in the process of computation.

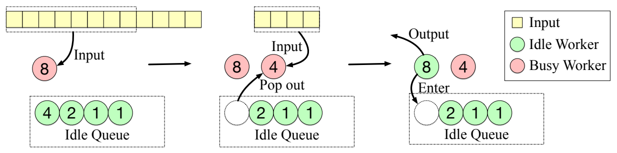


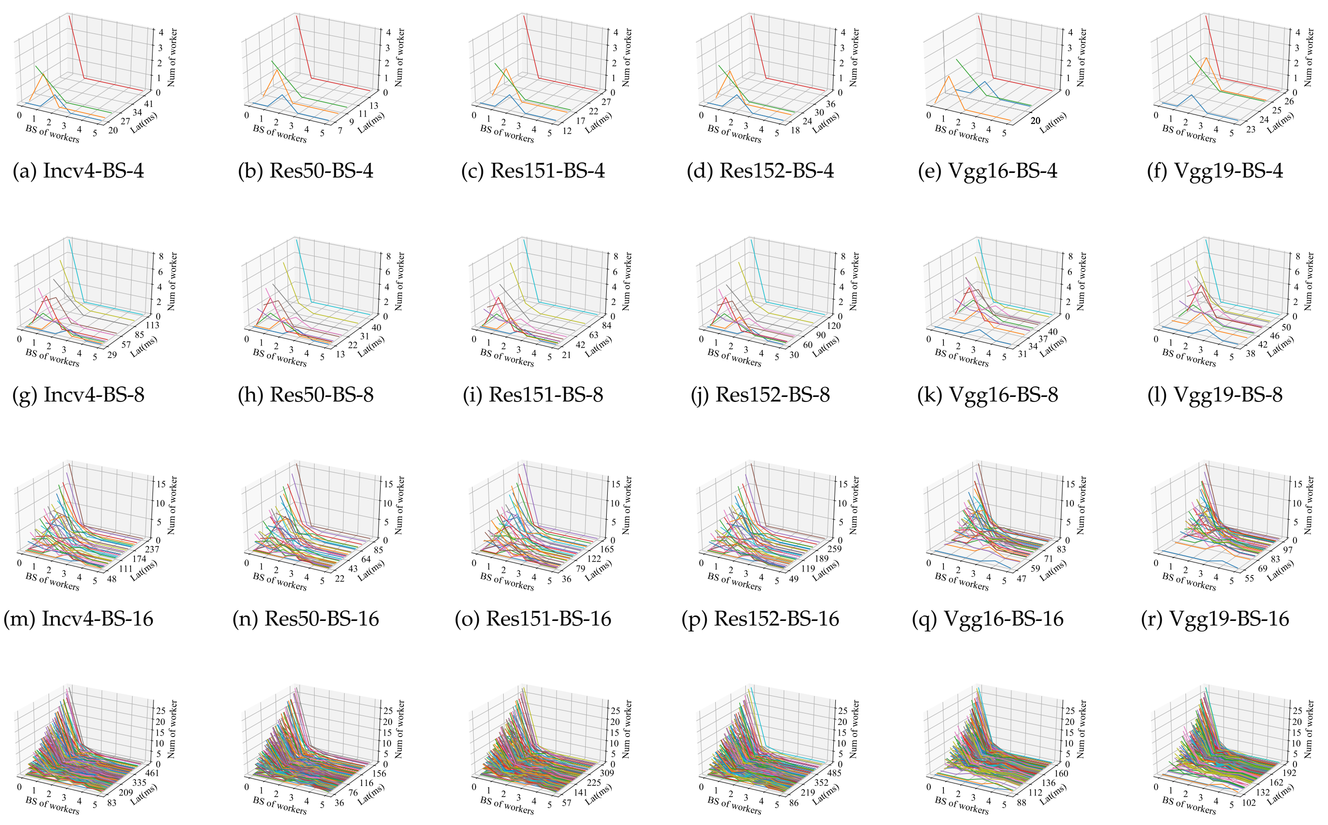
Fig. 10: Example of elastic batch scheduling.

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.

Algorithm 1 Naive Elastic batch scheduling algorithm.

```
Require: $\mathbf{N}, \mathbf{DevPtr}, \mathbf{Q}, \mathbf{MaxInf}, \mathbf{CurInf}$
1:  while True do
2:      if !$\mathbf{Q}.empty()$ then
3:          $\mathbf{R} \leftarrow \min(\mathbf{N}, \mathbf{MaxInf} - \mathbf{CurInf})$
4:          $\mathbf{Woker} \leftarrow \mathbf{Q}.front()$
5:          while $\mathbf{R} > 0$ and Worker do
6:              if $\mathbf{R} \geq \mathbf{Worker}.batchsize$ then
7:                  Schedule $\mathbf{DevPtr} \rightarrow \mathbf{Worker}.input$
8:                  $\mathbf{Worker}.run()$
9:                  $\mathbf{R} \leftarrow \mathbf{R} - \mathbf{Worker}.batchszie$
10:                 $\mathbf{DevPtr} \leftarrow \mathbf{DevPtr} + \mathbf{Worker}.batchsize$
11:                 $\mathbf{Q}.remove(\mathbf{Worker})$
12:             else
13:                 $\mathbf{Worker} \leftarrow \mathbf{Worker}.next()$
```

Figure 10 shows an example of how the elastic batch scheduler coordinates the memory pool and the inference engine work. 🟨 represents the concatenated input data in the memory pool. 🔴 represents the busy worker which are performing inference, while 🟢 represents the 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



(a) Incv4-BS-4 (b) Res50-BS-4 (c) Res151-BS-4 (d) Res152-BS-4 (e) Vgg16-BS-4 (f) Vgg19-BS-4

(g) Incv4-BS-8 (h) Res50-BS-8 (i) Res151-BS-8 (j) Res152-BS-8 (k) Vgg16-BS-8 (l) Vgg19-BS-8

(m) Incv4-BS-16 (n) Res50-BS-16 (o) Res151-BS-16 (p) Res152-BS-16 (q) Vgg16-BS-16 (r) Vgg19-BS-16

(s) Incv4-BS-32 (t) Res50-BS-32 (u) Res151-BS-32 (v) Res152-BS-32 (w) Vgg16-BS-32 (x) Vgg19-BS-32

Fig. 11: Profiling results of all mdoels under various loads.

to the load and GPUs operation status. There is a balance between the memory pool and the inference engine.

In summary, at the time of scheduling, we have to request a new idle worker of the multi-granularity inference engine to process the new batched requests. Because of the red-black tree used to sort the idle workers in the inference engine, the time complexity of requesting a new idle worker from the idle queue is $O(logM)$, where M is the number of the alive workers in the multi-granularity inference engine. And the time complexity of the rest steps is $O(1)$. Hence, the time complexity of the scheduling algorithm is $O(logM)$.

7.3 Enhanced Elastic Scheduling: E-Ebird

N-Ebird works under the guideline of the hypothesis, equal resource sharing. But resource contention of multiple workers on GPUs is difficult to figure out only with a hypothesis. Also, the configuration of the multi-granularity inference engine is fixed due to the limitation of global memory in the origin design of Ebird. After exploiting reusing memory techniques, E2bird is able to hold more workers in the multi-granularity inference engine. The configuration of the inference engine is flexible and able to adjust to load. A new question emerged: *what is the best combination of workers in the inference engine for a specific model under a specific load?* The deep learning models may require a specific batch size to be executed more efficiently due to the architecture of models and GPUs, which is neglected by N-Ebird.

Inspired by analysis of the elastic scheduling model in Section 7.1, we investigate the factual interaction of co-running inference workers by experiments to gain the more appropriate scheduling policy, E-Ebird. The work style of E-Ebird includes two phases: the offline phase and the online phase. In the offline phase, we profile all combinations of worker configurations under different load (TotaolBS) and those configurations with the best performance are recorded for guiding the online scheduling. Then in the online phase, the elastic batch scheduler will always use the best combination of workers for serving under different load, which has been profiled in the offline phase.

7.3.1 Interaction of Co-running Inference Workers

For a better understanding of the interaction among co-running inference workers, we profile the execution time performance of the multi-granularity inference engine. The configuration of each profiled inference engine is obtained by Equation 6. In Equation 6, i represents the num of workers with batch size $= 2^0$, j represents the num of workers with batch size $= 2^1$,...n represents the num of workers with batch size $= 2^5$. Given $TotalBS$, the summation of all the workers' batch size, we profile the inference engines which are configured with all the cases of $[i, j, k, l, m, n]$. Incidentally, we use the average latency of all workers in each case to denote the execution time performance instead of tail latency, which indicates better responsiveness.

$$TotalBS = 2^0 \times i + 2^1 \times j + 2^2 \times k + 2^3 \times l + 2^4 * \times m + 2^5 \times n \quad (6)$$

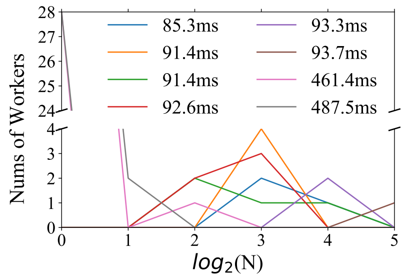


Fig. 12: Perfomance of multi-granularity inference engine for Resnet 50 with $TotalBS = 32$.

Figure 12 depicts the results of profiling inference engine with Res152 when setting $TotalBS$ at 32. Here, we only select the 6 cases with the shortest latency and 2 cases with the longest latency to make the results clear. In Figure 12, each line represents a case of the inference engine, the x-axis represents the binary logarithm of the worker batch size N, and the y-axis represents the num of each kind of worker. The legends in Figure 12 represent the corresponding latencies. As we can see, the execution time (85.3ms) of the inference engine with the configuration ($[0, 0, 0, 2, 1, 0]$) is the shortest. Meanwhile, the latencies in many cases are shorter than that ($93.7ms$) of the inference engine with a single worker whose batch size equals $TotalBS$. On the contrary, in the two cases, which consist of many workers whose batch size equals 1, the average latency can be 4 $\times$ of the shortest one.

Two key points can be concluded from the above observations: ❶ To a certain extent, multiple workers can achieve better responsiveness and higher throughput than a single worker; ❷ Too many workers with small batch sizes cause severe performance degradation, and the QoS target is be violated. Thus, under a specific load, there exists a specific configuration of the inference engine, which is neither a single worker with a large batch size nor too many workers with small batch sizes. An offline profiling helps the scheduler to seek it.

7.3.2 Offline Phase

Because the scheduler works under various loads, in the offline phase, E-Ebird profiles the performance of the inference engine under different loads. As illustrated in Table 1, workers with smaller batch sizes achieve better responsiveness but support lower throughput. Therefore, we set $TotalBS$ at $(4, 8, 16, 32)$, to simulate the various loads. We profiled all the models used in Section 8 for evaluation. The results are shown in Figure 11. In these sub-figures contained in Figure 11, the x-axis represents latency, the y-axis represents the binary logarithm of worker batch size, and the z-axis represents the num of each kind of worker.

After finishing profiling all the cases, E-Ebird selects the best inference engine configuration for each load according to the profiling results. All these configurations are saved as the scheduling guidelines. Then E-Ebird initiates the inference engine with a configuration that can satisfy all selected configurations. We still give an example of Res152, as shown in Table 4. The inference engine with the configuration ($[1, 1, 1, 2, 1, 0]$) can satisfy all selected scheduling

TABLE 4: Configuration of inference engine for Res152.

$TotalBS$	Configuration
4	[0,0,1,0,0,0]
8	[0,0,0,1,0,0]
16	[0,0,2,1,0,0]
32	[0,0,0,2,1,0]
Inference Engine	[1,1,1,2,1,0]

guidelines. It is worth noting that two workers of size 1 and size 2 are kept by default to support the extremely low load. There is no need for profiling the cases of ($TotalBS = 1, 2$).

7.3.3 Online Phase

Algorithm 2 Enhanced Elastic batch scheduling algorithm.

```
Require: N, DevPtr, Q, MaxInf, CurInf, MemState,
    EngineConfs, CurConf
 1: while True do
 2:     if MemState changed then
 3:         Update CurConf from EngineConfs
 4:         Update Q according CurConf
 5:     if !Q.$empty()$ then
 6:         R $\leftarrow \min(\mathbf{N}, \mathbf{MaxInf} - \mathbf{CurInf})$
 7:         Woker $\leftarrow$ Q.$front()$
 8:         while R $> 0$ and Worker do
 9:             if R $\geq$ Worker.$batchsize$ then
10:                 Schedule DevPtr $\rightarrow$ Worker.$input$
11:                 Worker.$run()$
12:                 R $\leftarrow$ R $-$ Worker.$batchszie$
13:                 DevPtr $\leftarrow$ DevPtr $+$ Worker.$batchsize$
14:                 Q.$remove$(Worker)
15:             else
16:                 Worker $\leftarrow$ Worker.$next()$
```

In the online phase, E-Ebird works under the guidelines got in the offline phase. Algorithm 2 lists how E-Ebird schedules the inference requests.

The most parameters used in Algorithm 2 are the same as that of N-Ebird, displayed in Table 3. Three new parameters are introduced, including **MemState**, **EngineConfs**, and **CurConf**. **MemState** represents the state of the memory pool, **EngineConfs** represents the configurations saved in the offline phase, and **CurConf** represents the current configuration of the inference engine. E-Ebird checks the **MemState** firstly before starting a new round of scheduling. The **MemState** denotes the load by the total number of pending requests in the GPU resident memory pool. If the load changes, the scheduler updates the **CurConf** guided by the **EngineConfs**. And then the scheduler updates the queue of idle workers accordingly. The remaining steps run as the same as the N-Ebird.

The extra steps for updating the configurations of the multi-granularity inference engine augment the complexity of scheduling policy. Upon changing the combination of alive workers, the queue of idle workers has to be reconstructed according to the batch size of workers, which is a process of building a red-black tree. Thus, under the worst case, the time complexity of the scheduling policy equals

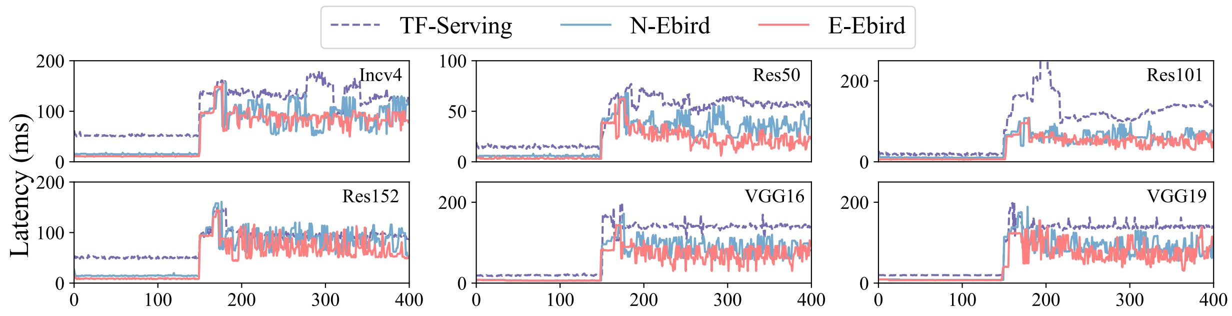


Fig. 13: The end-to-end latencies of the inferences in different benchmarks with TF-Serving, N-Ebird, and E-Ebird.

to $O(MlogM) + O(logM) = O(MlogM)$, where M is the number of the alive workers in the multi-granularity inference engine.

8 Evaluation

In this section, we first evaluate the effectiveness of E^{2}bird in improving the responsiveness and the throughput while satisfying the QoS requirement of deep learning-based services. We also dive into E^{2}bird for inspecting the hardware operating status and figure out the overhead of E^{2}bird. The two scheduling algorithms in E^{2}bird are both evaluated.

8.1 Experiment Setup

TABLE 5: 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)

We perform all the experiments on a machine equipped with the latest Nvidia Titan RTX GPU. The GPU has 72 SMs plus 576 Tensor cores and is able to deliver outstanding performance for deep learning inferences [43]. Table 5 lists the detailed experimental setup. As shown in Table 5, we use six widely-used deep neural networks as the online services to evaluate E^{2}bird. We compare E^{2}bird with SOTA deep learning serving system, TF-Serving, in the following of this section. TF-Serving uses the OPT_Wait policy described in Section 3 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.

8.2 Improving Responsiveness

In this experiment, we evaluate the effectiveness of E^{2}bird in improving the responsiveness of deep learning-based services with the 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 at a low rate, and the later 250 inferences are launched at a high rate. The load is high if the latencies of the inferences are close to the QoS target (200ms is used in this experiment) with TF-serving.

Figure 13 shows the end-to-end latencies of the inferences in different benchmarks when the inferences are served with TF-Serving, N-Ebird, and E-Ebird, respectively. As observed from this figure, both N-Ebird, and E-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, N-Ebird reduces the latency of the inferences ranging from 44.6% to 70.9% for the benchmarks. When the load is high, N-Ebird reduces the latency of the inferences ranging from 7.4% to 53.1% for the benchmarks. Under low load, E-Ebird maintains a similar latency performance with N-Ebird, which indicates they are all using workers with small batch sizes for serving. Under high load, E-Ebird achieves better performance ranging from 5% to 30% compared with N-Ebird.

The reason why the two algorithms can reduce the latency of the inferences at low load is that they reduce the unnecessary waiting time by the same strategy. Besides, they can improve the responsiveness at high load because they process 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 inputs and results are all transferred together, which all result in latency increase. Due to the delay of input transferring, the latencies of all inferences increase by the input transferring time. Meanwhile, all 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 N-Ebird and E-Ebird. In addition, because of the offline phase, E-Ebird adopts a better configuration of the inference engine than N-Ebird. Under high load, multiple workers of biggish batch sizes are activated in E-Ebird instead of these workers whose batch size equal 1 or 2 in N-Ebird, which avoids too many context switch and enables E-Ebird to work better.

Moreover, TF-Serving results in QoS violation of the inferences in Res101, when the load increases, which is mainly because TF-Serving processes batches of inferences

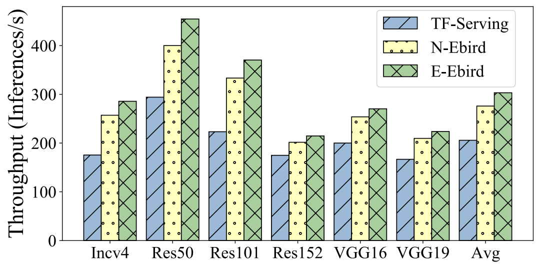


Fig. 14: The inference processing throughput of the benchmarks with TF-Serving, N-Ebird, and E-Ebird while guaranteeing the QoS.

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, N-Ebird and E-Ebird can process the bursty inferences if the current inferences are not able to utilize the GPU fully. They can always guarantee the QoS of deep learning-based services no matter the load is bursty or not.

8.3 Increasing Throughput while Guaranteeing the QoS

In this subsection, we evaluate E^{2}bird 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 14 presents the achieved inference processing throughput with TF-Serving, N-Ebird, and E-Ebird, while the latencies of the inferences are shorter than the QoS target. As we can see from this figure, N-Ebird and E-Ebird improve the inference throughput of all the benchmarks compared with TF-Serving. On average, N-Ebird improves the throughput by 34.4% compared with TF-Serving and E-Ebird additionally improves the throughput by 9.9% compared with N-Ebird. In this way, given the same peak load of a deep learning-based service, fewer GPUs are needed to host the service with N-Ebird and E-Ebird.

The two algorithms in E^{2}bird are able to improve the throughput while guaranteeing the QoS because they overlap data transfer and computation. N-Ebird, and E-Ebird eliminate the long GPU idle time due to data transfer. On the contrary, the SMs of the GPU in TF-Serving are idle when the input data/the inference result is transferred to/from the GPU. E-Ebird achieves a higher utilization of GPU resources than N-Ebird, which mainly benefits from offline profiling.

As shown in Figure 14, the throughput improvements are big 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. The benefit of overlapping data transfer and computation declines if the data transfer takes a large percentage of an inference's end-to-end latency.

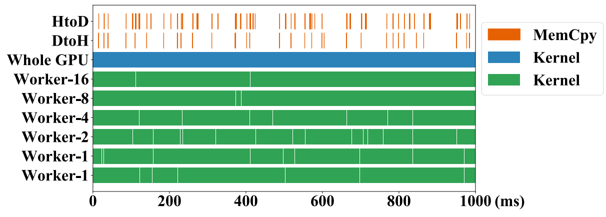


Fig. 15: Snapshot of inference processing with N-Ebird.

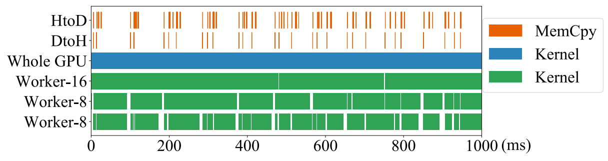


Fig. 16: Snapshot of inference processing with E-Ebird.

8.4 Diving into E^{2}bird

To better understand why E^{2}bird performs better than TF-Serving, Figure 15 shows the execution trace of executing inferences of Res152 with N-Ebird, and Figure 16 shows that with E-Ebird (Figure 3 shows a similar trace with TF-Serving). In these two Figures, "Worker-n" shows the worker's kernel execution for inference batches of size n, "Whole GPU" shows all the kernel execution in all the workers on the whole GPU.

Comparing Figure 15 and Figure 3, the inputs of inferences are transferred to GPU separately in N-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 15 (Row "Whole GPU").

As observed from Figure 15, 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 the worker with the smaller batch size is also sparser than that of the worker with a larger batch size, indicating that N-Ebird intends to schedule a worker with the larger batch size within the idle worker queue 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 N-Ebird is able to reduce the end-to-end latency of inferences at high load, as shown in Figure 13.

Comparing Figure 15 and Figure 16, the workers that E-Ebird uses are less than N-Ebird. Though E-Ebird owns many workers of different batch sizes in it, E-Ebird always chooses the best inferences engine configuration according to profiling results under a specific load. As shown in Figure 12, E-Ebird only uses 3 workers whose batch sizes are $(8, 8, 16)$ under the high load. Under the same conditions of the time interval, "Row Whole GPU" of E-Ebird is also denser than that of N-Ebird, which indicates a higher utiliza-

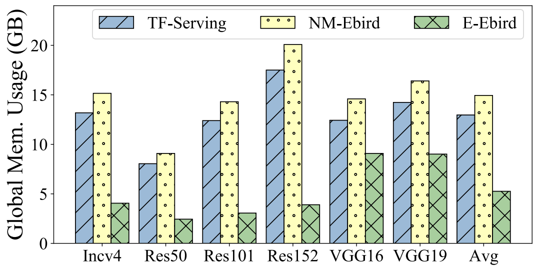


Fig. 17: Global memory usage of TF-Serving, NM-Ebird, and E^{2}bird.

tion of E-Ebird. This phenomenon also explains why E-Ebird achieves lower latency under high load and supports a higher throughput. The fixed inference engine configuration of N-Ebird causes the performance decline.

8.5 Overhead of E^{2}bird

The performance overhead of the memory pool and inference engine has been analyzed when validating their design philosophy in Section 5.3 and Section 6.3. The rest overhead of E^{2}bird comes from the multi-granularity inference engine owing to maintaining multiple inference workers.

Figure 17 shows the global memory usage of TF-serving, NM-Ebird, and E^{2}bird. **Here NM-Ebird represents the E^{2}bird without the static memory optimizations**. As we can see, NM-Ebird uses 15.2% more global memory space compared with TF-Serving. NM-Ebird uses more global memory space because workers duplicate the global memory used for storing the weight and intermediate results of the deep learning network . Moreover, the extra global memory [44] needed by convolution is also duplicated. After adopting optimizations, E^{2}bird reduce the global memory consumption dramatically. On average, E^{2}bird now reduce the global memory usage by 64.8% compared with NM-Ebird and 59.4% compared with TF-Serving.

9 Conclusion

E^{2}bird improves responsiveness and throughput for deploying deep learning services in datacenters outfitted with GPUs. For these purposes, E^{2}bird enables the GPU-side prefetch mechanism and the elastic batch scheduling policy for the deep learning serving system. As far as we know, E^{2}bird is the first GPU-side batching system for deep learning serving system on GPUs. Through comparing the performance of E^{2}bird and TF-Serving (State-of-the-art deep learning serving system), we verify the effectiveness of E^{2}bird in eliminating the waiting time for responsiveness and overlapping data transfer and computation for GPUs when providing deep learning services. Generally, E^{2}bird enhances responsiveness. Moreover, E^{2}bird improves the throughput by 47.4% on average compared with state-of-the-art solutions, TF-Serving.

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] 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.
[12] 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.
[13] 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.
[14] Multi-process service. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[15] 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.
[16] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 269–281.
[17] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 163–174.
[18] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.
[19] 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.
[20] 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.
[21] 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.
[22] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: Qos for accelerated machine learning systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 172–184.
[23] 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.

[24] 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.

[25] 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.

[26] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: a gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[27] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Inferline: Ml inference pipeline composition framework," *arXiv preprint arXiv:1812.01776*, 2018.

[28] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

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

[30] 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.

[31] 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.

[32] 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.

[33] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[34] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[35] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," *arXiv preprint arXiv:1902.06822*, 2019.

[36] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[37] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," 2011.

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

[39] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 41–53.

[40] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[41] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[42] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," 2006.

[43] "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

[44] 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.

Weihao Cui received his B.Sc. degree from Shanghai Jiao Tong University, China. He is currently an Ph.D. student in the field of computer science under supervision of Dr. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. His research interests include high performance computing and resource management of accelerators in datacenters.

Quan Chen is a tenure-track associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include High performance computing, Task Scheduling in various architectures, Resource management in Datacenter, Runtime System and Operating System. He got his Ph.D. degree at June 2014 from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.

Han Zhao received his B.Sc. degree from Shanghai Jiao Tong University, China. He is currently an Ph.D. student in the field of computer science under supervision of Dr. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. His research interests include high performance computing and resource management of accelerators in datacenters.

Mengze Wei received her B.Sc. degree from Shanghai Jiao Tong University, China. She is currently an M.Sc. student in the field of computer science under supervision of Dr. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. Her research interests include high performance computing and resource management in Datacenter.

Minyi Guo Guo received the Ph.D. degree in computer science from the University of Tsukuba, Japan. He is currently Zhiyuan Chair professor and head of the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, pervasive computing, big data and cloud computing. He is now on the editorial board of IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing and Journal of Parallel and Distributed Computing. Dr. Guo is a fellow of IEEE, and a fellow of CCF.