DLBooster: Boosting End-to-End Deep Learning Workflows with Offloading Data Preprocessing Pipelines

Yang Cheng
cheng-y16@mails.tsinghua.edu.cn
Tsinghua University
Microsoft Research

Dan Li
tolidan@tsinghua.edu.cn
Tsinghua University

Zhiyuan Guo
v-zhguo@microsoft.com
Microsoft Research
Beihang University

Binyao Jiang
v-bijian@microsoft.com
Microsoft Research
Shanghai Jiao Tong University

Jiaxin Lin
v-jiaxl@microsoft.com
Microsoft Research
Beihang University

Xi Fan
v-xif@microsoft.com
Microsoft Research
Shanghai Jiao Tong University

Jinkun Geng
steam1994@163.com
Tsinghua University

Xinyi Yu
v-xinyyu@microsoft.com
Microsoft Research
Shanghai Jiao Tong University

Wei Bai
webai@microsoft.com
Microsoft Research

Lei Qu
Lei.Qu@microsoft.com
Microsoft Research

Ran Shu
Ran.Shu@microsoft.com
Microsoft Research

Peng Cheng
pengc@microsoft.com
Microsoft Research

Yongqiang Xiong
yongqiang.xiong@microsoft.com
Microsoft Research

Jianping Wu
jianping@cernet.edu.cn
Tsinghua University

ABSTRACT

In recent years, deep learning (DL) has prospered again due to improvements in both computing and learning theory. Emerging studies mostly focus on the acceleration of refining DL models but ignore data preprocessing issues. However, data preprocessing can significantly affect the overall performance of end-to-end DL workflows. Our studies on several image DL workloads show that existing preprocessing backends are quite inefficient: they either perform poorly in throughput (30% degradation) or burn too many (>10) CPU cores. Based on these observations, we propose DLBooster, a high-performance data preprocessing pipeline that selectively offloads key workloads to FPGAs, to fit the stringent demands on data preprocessing for cutting-edge DL applications. Our testbed experiments show that, compared with the existing baselines, DLBooster can achieve 1.35×∼2.4× image processing throughput in several DL workloads, but consumes only 1/10 CPU cores. Besides, it also reduces the latency by 1/3 in online image inference.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337892

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**.

KEYWORDS

Deep learning, data preprocessing, cloud computing, FPGAs

ACM Reference Format:
Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jiaxin Lin, Xi Fan, Jinkun Geng, Xinyi Yu, Wei Bai, Lei Qu, Ran Shu, Peng Cheng, Yongqiang Xiong, and Jianping Wu. 2019. DLBooster: Boosting End-to-End Deep Learning Workflows with Offloading Data Preprocessing Pipelines. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3337821.3337892

1 INTRODUCTION

Today, DL is prevalent due to its recent substantial progress in many areas, such as image classification [26, 35, 36], speech recognition [18], recommender system [45], and so on. As one of the largest cloud service providers in the world, we have witnessed various kinds of DL tasks submitted by our customers during the past few years. Given the prevalence of DL, much effort has been made to accelerate DL workloads, especially in the cloud. For example, advanced DL frameworks such as TenforFlow [1], PyTorch [34], and MXNet [6] have been proposed and widely supported by today's public cloud providers as a service. Meanwhile, computation and communication hardware have also been developed to provide infrastructural support for DL acceleration. For example, NVIDIA Tesla V100 and Google TPU-v3 become available in the cloud, which

can greatly improve the computation speed for heavy DL workload. RDMA and FPGA NICs are also emerging to offload the packet processing logic to hardware and achieve higher throughput, as well as ultra-low latency [22, 23, 25, 28, 40, 48].

Despite the aforementioned efforts [3, 7, 8, 23, 24, 32, 33, 47], we notice that most of them focus on the DL acceleration through the training/inference process, but ignore the overheads of data preprocessing. Actually, data preprocessing serves as a pre-conditional step for training/inference and can seriously affect the overall performance for DL training/inference. Moreover, with more powerful computing hardware and higher network capacity, the training/inference bottlenecks are mitigated to a great extent [22, 23, 46], leaving data preprocessing as an increasingly distinct bottleneck in today's DL workflows. For example, while training AlexNet [35] on a GPU cluster, we have the following observations: (1) Caffe [31] using the LMDB [50] backend decreases the training throughput by ~30%, compared with training with synthetic data; and (2) Caffe, which parses data using the CPU at runtime, achieves only ~25% training performance in the default configuration or makes up the performance gaps by burning more than 12 CPU cores per GPU.

Burning CPU cores seems to be a straightforward choice to improve the preprocessing speed and match the GPU power. However, this approach is inefficient and even infeasible due to general concerns such as performance degradation and poor scalability. Particularly, when deploying DL in the cloud, burning CPU cores will bring the expensive cost to both users (CPUs are charged in the cloud) and cloud providers (high power consumption increases their maintenance costs). In addition to burning CPU cores, there are many other data preprocessing backends widely used today. However, from our studies, they also suffer from several limitations such as downgraded performance, expensive cost (economics and time), scalability, and lack of generality, which we discuss in detail in Section 2.2. Unfortunately, these limitations widely exist in today's DL frameworks but are not well addressed, which make them the bottlenecks in modern DL deployments, especially in the cloud.

Considering the deficiencies with existing works on data preprocessing, we propose DLBooster to make DL great again. DLBooster is an online data preprocessing backend to speed up end-to-end DL workflows, which delivers an online decoding service by offloading the data preprocessing workloads to FPGAs, and can simultaneously feed the processed data into several GPU engines. However, DLBooster still faces the following challenges:

(1) **Co-design with hardware and software:** Different from software systems, DLBooster is a system which needs to consider the co-existence of software and hardware. We need to communicate with the decoder in FPGA from user space and *drive* it to work correctly. Moreover, data access in OS and FPGA differs greatly, which requires us to design a high-performance I/O interface for the decoder in FPGA. We discuss them in depth in Section 3.4.1 and Section 3.4.2.

(2) **Balance between workload and resource constraint:** It is very inefficient to naively offload all data preprocessing workloads to the FPGA device, mainly due to the constraints of FPGA hardware resources, and we discuss our considerations about what workloads should be offloaded to FPGAs and how to pipeline them efficiently in Section 3.3.

(3) **Compatibility to different DL engines and applications**: DLBooster should not only achieve high performance, but should also be open to different DL engines and different DL applications. We discuss the design of the DLBooster interface in Section 4.1.

Having addressed the above challenges carefully, we build a DLBooster prototype and integrate it with two popular DL frameworks, namely, NVCaffe [13] and TensorRT [15]; then, we evaluate the performance of DLBooster on two representative end-to-end image DL workflows, namely, offline training and online inference. Our results show that DLBooster improves the image processing throughput by up to 1.35×~2.4× and reduces the image processing time by 1/3. This throughput in certain cases comes close to the performance bound of GPU.

One may concern about the programming with FPGAs and the cost of FPGA devices when using DLBooster. However, we show the investment in DLBooster is worthwhile (in Section 5.4), and we are trying to minimize users' effort on using DLBooster by our careful designs and continual optimization. In summary, this paper makes the following contributions.

- We first address the importance of data preprocessing for today's DL systems and conduct comprehensive studies on its limitations.
- We design an online data preprocessing backend by selectively offloading decoding workloads to FPGAs to speed up DL workflows.
- With careful design, we build a DLBooster prototype and prove its efficiency in several end-to-end DL workflows of image applications.
- We expose friendly APIs and pluggable FPGA decoders, so that common users can easily migrate DLBooster to different DL engines for the performance boost.

The rest of this paper is organized as follows: Section 2 illustrates the full stack of the DL workflow in detail and analyzes the limitations of existing data preprocessing backends, which motivates us to design DLBooster in Section 3. Section 4 presents the implementation of DLBooster, and Section 5 demonstrates the potential of DLBooster based on several image DL workflows. We show the related work in Section 6 and conclude this work in Section 7.

2 BACKGROUND AND MOTIVATION

2.1 DL Workflow

DL [39] is a class of machine learning algorithms, which represents complex tasks with a large number of connected neurons. DL greatly simplifies the complexity of models while maintaining high accuracy and has been widely used to solve many cutting-edge tasks today. Particularly, an end-to-end DL workflow (shown in Figure 1) consists of several phases, which are listed as follows.

Data collection is the first step of an end-to-end DL workflow and is used to prepare data for learning. Usually, DL users prepare data from historical samples generated by themselves or public datasets, such as ImageNet[41], MJSynth[29], AudioSet[21], etc. Data collection can greatly affect the accuracy of DL applications.

Data preprocessing transforms data in various formats to the unified input for DL models and varies greatly for different DL applications. In general, data preprocessing consists of two steps, namely, data standardization which extracts features and reshapes

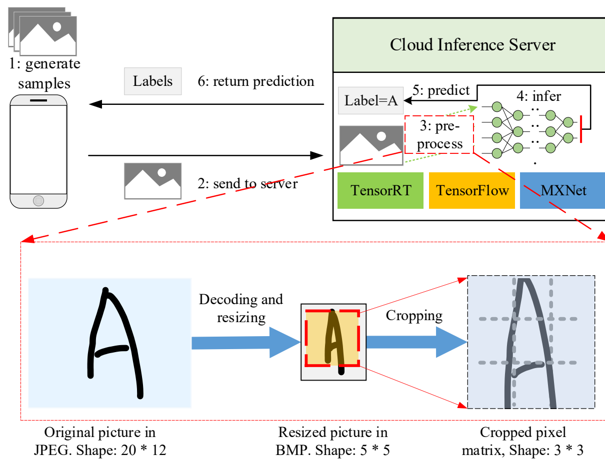


Figure 1: The full stack of a DL workflow. Taking the cloud-based online inference as an example, the client on device first generates data and sends the data to the server on cloud, and the inference engine on the server then makes prediction based on the data and returns the labels to the client.

them to match the input of a DL model, and data augmentation which adopts augmentation technologies such as scaling, cropping, and rotation, to avoid *overfitting* when learning from a small dataset.

Training/inference is the core part of an end-to-end DL workflow. (1) Training is used to optimize a model (usually with stochastic gradient descent (SGD) [5]) based on history data, which usually consists of thousands of iterations[1]. Training a DL model [4, 49] with SGD (in one iteration) comprises two steps, namely, *forward* pass which is used to give predicted values for input samples, and *back-propagation* pass which is used to compute gradients by propagating the errors[2] layer by layer in a reverse order. After that, the model's parameters are refined by their gradients[3]. (2) Inference is used to predict the unknown data using the well-tuned model. Inference on a DL model de facto executes a series of multilayer *forward* functions. Both training and inference are computation-intensive workloads and are popular in the cloud (with powerful GPUs) today.

Among them, data preprocessing is a fundamental step in a DL workflow, which needs to tackle heterogeneous data formats according to different DL tasks. For example, as for image learning tasks, image samples in various formats (e.g., JPEG, PNG.) are decoded to extract the pixel matrices. As for speech learning tasks, audio samples undergo a discrete cosine transform to obtain the spectra data. In languages learning workflows, text samples in different languages are quantized to obtain the vectorized features.

Considering the huge volume of data to process, data preprocessing is computation-intensive and time-consuming. Reviewing the existing DL frameworks, there are two types of serving primitives for data preprocessing, which we categorize as *offline primitive* and *online primitive*. The *offline primitive* requires great efforts to process dataset in advance and offers services by loading the processed data from locally stored disks or databases (DBs). By contrast, the *online primitive* offers data preprocessing services by decoding at runtime. Data preprocessing backends such as LMDB [50] in Caffe, TFRecord [17] in TensorFlow, and RecordIO [2] in MXNet offer offline preprocessing primitives, but most DL frameworks offer online preprocessing primitives by burning CPU cores or GPU cores.

[1]This process is also known as the *learning*.
[2]The differences between the predicted values of *forward* pass and the ground truth.
[3]Today, DL models are usually trained among distributed clusters, where the gradients will be synchronized with each other before they are applied to the model's parameters.

2.2 Limitations of Existing Data Preprocessing Backends

Unfortunately, the aforementioned data preprocessing backends work inefficiently on existing GPU clusters, and the situation will be more serious in the future where GPUs become faster. The limitations of existing backends can be summarized as follows.

(1) **Performance degradation:** Performance degradation can appear in both online and offline data preprocessing backends. As shown in Figure 2, when training AlexNet by Caffe in a GPU cluster, the LMDB-enabled Caffe downgrades GPU performance by 30% due to the competition on the shared DB backend as more GPUs are used, and the CPU-based Caffe achieves only 25% of the GPU performance under the default configuration. Besides, backends such as nvJPEG [16] which exploits GPUs to preprocess data, will compete with model computation for GPU cores in an end-to-end DL workflow. In our observations, the nvJPEG can dominate 40% GPU utilization in several inference workflows, thus downgrading the GPU performance in model computation by more than 30%.

(2) **Expensive cost:** Existing data preprocessing backends are expensive in terms of both time and economics. First, offline backends such as LMDB require significant time to parse data first before training a model. For example, we spent more than 2 hours to prepare the LMDB backend for ILSVRC12 [41].[4] The significant time cost hurts the benefit of users who move DL workloads to the cloud where VMs are charged in terms of time cost. Second, online backends usually offer high-performance online preprocessing services by burning CPU/GPU cores, which bring the significant cost to both users and cloud providers for the following reasons: (a) CPUs and GPUs are expensively charged in the cloud or market; (b) The significant power consumption increases the cost of maintenance.

(3) **Scalability:** Today, one GPU server can hold up to 8 high-performance GPUs and two CPUs (up to 48 cores in all). However, according to our observations, the NVIDIA Tesla V100 can process 5,000 images per second when inferring the ResNet-50 model whereas each Xeon E5 CPU core can decode only 300 images per second, and the demands on CPU cores to fully boost GPUs' performance have already exceeded what such servers can offer. Such situations can be severer in GPU clusters such as NVIDIA DGX-2 [14], where 16 Tesla V100 GPUs and 2 CPUs (48 cores in all) are equipped, and each GPU can use at most 3 cores on average. Therefore, it can be implied that the number of CPU cores limits the scalability of the DL workflow when more GPUs are used.

Our motivation: Based on the aforementioned analysis, we can conclude that offline data preprocessing backends are limited by their performance, time cost and compatibility, whereas online backends are limited by their scalability and economic cost. Both of

[4] Recent works [3, 8, 24, 47] report that it is possible to train the AlexNet model in minutes, but we find that they only use synthetic datasets and skips the data preprocessing step, which is not impractical for industrial DL workflows.

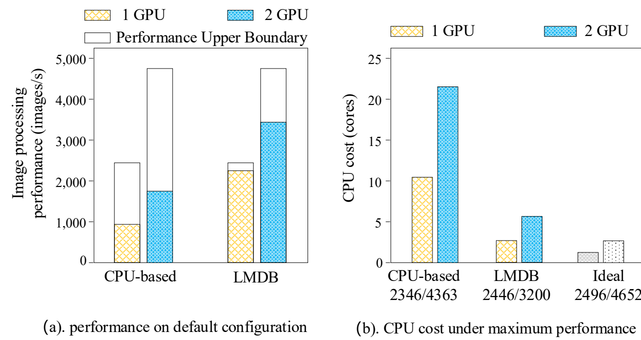


Figure 2: Statistical results of training AlexNet on NVIDIA P100 with the Caffe engine under data parallelism. The batch size is set to 256 images/GPU.

them are becoming bottlenecks in end-to-end DL workflows, and such situations will deteriorate considering the rapid development of GPUs in the future. These limitations in existing preprocessing backends also inspire us to design DLBooster, which aims to offload some key workloads to FPGAs, thus reducing the burden for CPU cores and effectively accelerate the DL preprocessing.

3 DLBOOSTER DESIGN

In this section, we first describe our design principles for DLBooster. Then, we summarize the challenges in designing such a data preprocessing backend. With these constraints and challenges, we start our roadmap to build the prototype of DLBooster, which offers high-performance online services by selectively offloading data preprocessing to the FPGA device.

3.1 Design Principles

There are many options for building an end-to-end AI acceleration solution across hardware and software. We start with the following goals and constraints for building our DLBooster.

Offering hybrid data preprocessing service: Considering the distinct limitations of offline backends in Section 2.2, we choose to design DLBooster as an online data preprocessing backend to serve more DL workflows. However, DLBooster can also cache the static dataset for iterative learning workflows as the offline backends. Particularly, DLBooster preprocesses all data in the first epoch and caches them in memory as it can. After that, DLBooster loads the processed data from the memory cache in the following epochs.

Offloading decoding logic to FPGAs: Compared to CPU-based and GPU-based backends that offer online preprocessing services, DLBooster exploits an FPGA device to decode online instead of burning CPU cores or GPU cores for the following reasons: (1) The CPU-based backend can scale poorly when consuming too many CPU cores that are supposed to process other workloads (e.g., parameter aggregation of parameter server (PS)). (2) GPU-based backends such as nvJPEG will downgrade the performance of model computation when competing on GPU cores in an end-to-end DL workflow. (3) FPGAs not only can achieve high performance and scale well in decoding workloads but also run at a lower cost.

However, instead of offloading all preprocessing workloads to FPGA devices, we selectively offload workloads that can be executed effectively by FPGAs to balance the cost and gains. For example, in designing image preprocessing decoders, we offload the decoding and the resizing to FPGAs and leave the data augmentation to GPU to achieve maximum performance at the lowest cost.

Maintaining programming flexibility: Programmability is always a concern for large-scale FPGA deployments. As a general preprocessing backend for different DL workloads and different engines, DLBooster considers programming flexibility during design and offers open interfaces, to benefit more AI workflows as follows: (1) On the FPGA side, DLBooster allows programmers to redesign their decoder on OpenCL [43], and the decoder in FPGA is pluggable, which allows users to download relevant preprocessing mirrors to FPGA devices for different applications (e.g., language models, video models and speech models). (2) In the software layer, DLBooster isolates each data preprocessing backend from the others and offers friendly interfaces for users to allow them to implement the adaption for different workloads with less effort. (3) DLBooster decouples the complex data preprocessing workloads from compute engines to flexibly adapt to different DL frameworks (e.g., Caffe, TensorRT). With simple interfaces exposed by DLBooster, users can easily integrate it with different DL libraries in their production.

3.2 DLBooster Architecture

The architecture of DLBooster is shown in Figure 3. DLBooster co-designs hardware and software. In the logic view, DLBooster can be divided into the data plane and the control plane, and the control plane further includes three main components, namely, FPGA decoder, host bridger and compute engine. We will introduce them in a bottom-up order.

The data plane is located in the first layer of DLBooster. It retrieves the raw data from both the local disk and the Internet (NIC), considering different DL workflows. Then, the data plane will feed the raw data into the backend for further processing by the FPGA decoder and host bridger.

The FPGA decoder drives the FPGA device to execute data preprocessing. It retrieves the data fed by the data plane and submits the processed output to the host bridger in the user space. Besides, the decoder is designed as a pluggable component and allows the programmer to define the decoding logic for different DL workloads.

The host bridger is the core component of the whole system; it connects the decoder in FPGA with the compute engine in GPU. There are three daemon threads launched in the host bridger, i.e., FPGA handler, GPU handler, and dispatcher. Specifically, the FPGA handler offers abstraction and interfaces with the FPGA decoder, whereas the GPU handler offers the abstraction and interfaces with the GPU devices in the compute engine. The dispatcher coordinates with two handlers, and moves the processed data from the decoder to the compute engine using a recycled memory pool.

The compute engine is located in the top layer of DLBooster, and it manages the training/inference logic in GPU space. Within the compute engine, every GPU device is isolated from the others and fetches data from its individual pipeline connected with the dispatcher in the host bridger.

We connect all the neighboring components with high-speed channels, so that they can be cooperated to offer the high-performance online data preprocessing service.

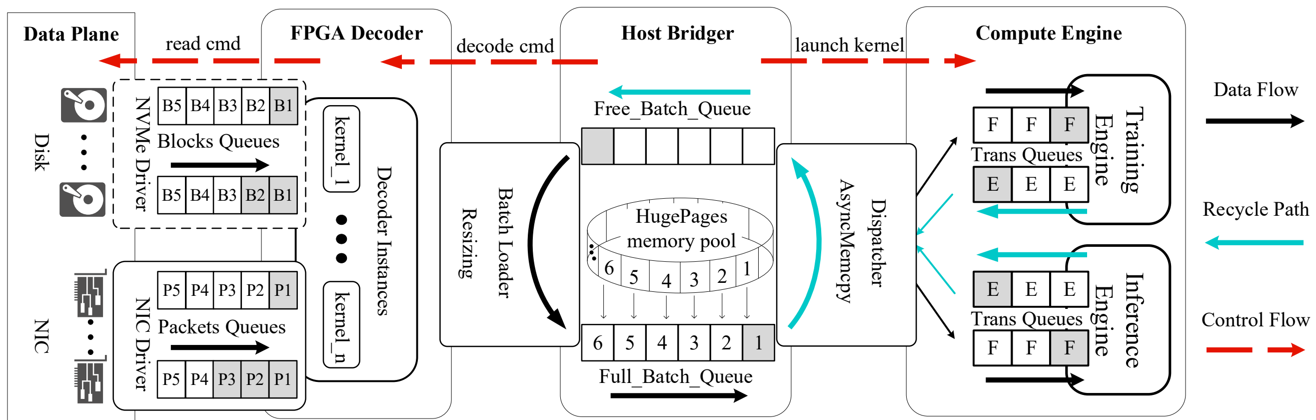


Figure 3: DLBooster architecture. There are 4 control planes used to manage multiply devices in different layers, and each control plane coordinates with its neighbors. The data flow is: disk/NIC → FPGA decoder → host memory → device memory.

3.3 FPGA Decoder Design

DLBooster designs a decoder in FPGA to address the preprocessing job, and the running logic in the decoder can be replaced in different DL workloads. We take image processing as an example to illustrate the decoder design (shown in Figure 4).

At the top of the decoder, the *parser* receives *cmds* from the host bridger through a *FIFO queue*, decodes these *cmds* to extract metadata, and then fetches real data from disk or memory via the *DataReader*. Meanwhile, it records the memory address to hold the processed results via the memory management unit (MMU). Following that, the fetched data are sent to the Huffman decoding unit, which is connected to the inverse discrete cosine transform (iDCT) unit to recover the original pixel matrix. After decoding, the pixel matrix is passed to the resizing unit to reshape its final outputs, and is then written to the host memory via DMA. Finally, a FINISH signal is triggered to notify the host bridger.

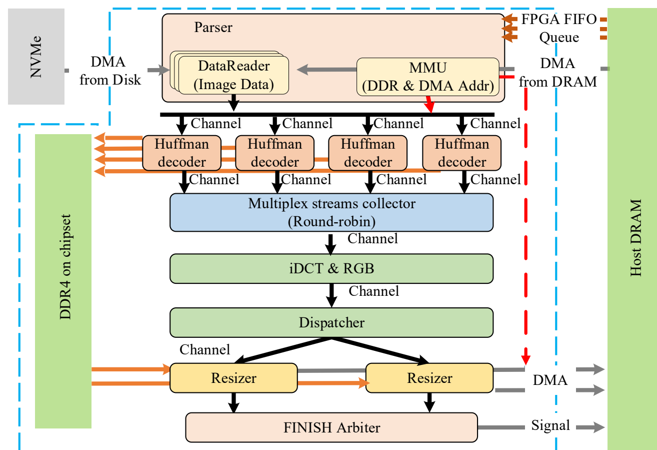


Figure 4: FPGA-based decoder architecture, with image processing as an example to show the details

We employ two steps to further optimize the decoder.

(1) We decouple the processing logic into three units (i.e., Huffman decoding unit, iDCT unit, and resizing unit) instead of merging them into one, which allows each of them to work in pipelining and increases the parallelism.

(2) As for each unit, we can flexibly scale running logic to different numbers of configurable logic blocks (CLB) in FPGA, according to its workloads and hardware constraints. In this way, the three processing units can achieve better load balance, and none of them will become the straggler that prolongs the overall processing time. For example, we implement a 4-way Huffman decoding unit and a 2-way resizing unit to achieve higher performance when decoding images.

3.4 Host Bridger Design

In order to simplify the processing logic and better coordinate the decoder and compute engine, the host bridger offers two types of abstraction in DLBooster, namely, decoding abstraction and memory managing abstraction.

3.4.1 Decoding Abstraction. The hardware devices can be heterogeneous, and a unified management is desirable. To provide easy programmability and mask the low-level hardware operation to common users, DLBooster sets up an abstraction, namely, *FPGAReader* (run as Algorithm 1), to offer online data preprocessing services. Generally, it pushes *cmds* to the FPGA decoder and synchronizes the processed data stream to the compute engine.

In *FPGAReader*, *FPGAChannel* is set up to serve as an abstraction interacting with the FPGA decoder. Each *FPGAChannel* is bound to one FPGA decoder and works independently. Moreover, a *DataCollector* is set up as a data abstraction, which translates the metadata (i.e., block information) that describes the storage information of the data on the disk or generates the metadata (i.e., physical address of memory) that describes where the data are placed by NICs. The *DataCollector* is globally shared by its callers in generating *cmds* for FPGA decoders.

FPGAReader works in an asynchronous manner: it aggressively submits *cmds* to the FPGA decoder through the *FPGA FIFO queue* maintained by *FPGAChannel* and pulls the processing status with the best effort. Such an asynchronous design enables *FPGAReader* to achieve high throughput and keep low latency.

3.4.2 Memory Managing Abstraction. Since the FPGA decoder cannot operate on the virtual memory, we need to enable a mapping mechanism (e.g., *mmap*) in DLBooster, which maps memory address between the virtual space and the physical space, to facilitate the operation of the decoder in FPGA and the host bridger in user space.

Algorithm 1: Asynchronous FPGAReader

```
   Input: file_manifest, memory_pool
   Output: None
1  data_collector ← BuildChannel (data_channel);
2  fpga_channel ← FPGAInit (Queue_ID);
3  Loops_entry:
4  foreach file in data_collector do
5      mem_hoder ← free_batch_queue.peak() ;
6      if mem_hoder is NOT_AVAILABLE then
7          mem_carriers ← fpga_channel.drain_out();
           foreach mem in mem_carriers do
8              full_batch_queue.push(mem);
9          mem_hoder ← free_batch_queue.pop() ;
10     f_metainfo ← file.get_metainfo() ;
11     cmd ← cmd_generator(f_metainfo,
        mem_hoder.phyaddr()+offset);
12     mem_carriers ← fpga_channel.submit_cmd(cmd);
13     foreach mem in mem_carriers do
14         full_batch_queue.push(mem);
15 if running then
16     goto Loops_entry;
17 file_pool.recycle();
18 fpga_channel.recycle();
19 return;
```

However, in today's DL applications, data are usually preprocessed in batches, and the required size of contiguous memory has exceeded what *mmap* can offer. To avoid the overhead introduced by copying small pieces, DLBooster designs its memory mapping mechanism (shown as Algorithm 2) based on the Linux *HugePage*, which takes charge of the whole control over memory management.

More concretely, DLBooster designs a *memory pool* to maintain the allocated memory and offers memory accessing abstraction via two queues, namely, *Full_Batch_Queue* and *Free_Batch_Queue*. At the start-up, DLBooster allocates a very large amount of memory ($>$ 1GB) in continuous space and slices it into smaller pieces. The small memory pieces are inserted into *Free_Batch_Queue* for later use. Each small memory piece contains several items, such as physical address, virtual address and memory size, to record its identification.

During the data preprocessing, *FPGAReader* fetches a memory unit from the *Free_Batch_Queue* and encapsulate its physical address (with an offset that current data are placed in the batch) into *cmds*. Since it works in an asynchronous mode, *FPGAReader* may aggressively encapsulate the data to process into the memory units with best effort, and soon run out of the memory for new data. To avoid this, *Free_Batch_Queue* will be blocked until a new memory unit is available. After receiving the signal from the FPGA decoder, *FPGAReader* puts the memory unit that carries a batch of data to the *Full_Batch_Queue*.

On the other side of the two queues, *Dispatcher* fetches a memory unit from the *Full_Batch_Queue*, synchronizes the processed data to compute engines, and then inserts the memory unit to the *Free_Batch_Queue* for future use. Those two queues connect *FPGAReader* and *Dispatcher* and allow data to *flow* from the FPGA decoder to the compute engine at high speed.

Algorithm 2: HugePage Memory Managing

```
   Input: size,counts
   Output: None
1  base_addr ← get_HugePage (size * counts);
2  for index = 0; index < counts; index++ do //pre-allocate
3      items.phy_addr ← base_addr + index * size ;
4      items.virt_addr ← phy2virt (items.phy_addr);
5      free_batch_queue.push (items);
6  MemMagager.insert(free_batch_queue);
7  MemMagager.insert(full_batch_queue);
8  Return MemMagager;
```

3.4.3 Dispatcher Design. Most DL frameworks allow multiple GPU devices to jointly work with data parallelism. Different from CPU cores, the GPU device cannot operate on the host memory or share its memory with other devices. To enable batched data to flow from the host space to the GPU space, the host bridger leverages a *Dispatcher* to dispatch the processed data. The running logic of the *dispatcher* is shown in Algorithm 3.

During the initial stage, all compute engines will register their communication channels (i.e., *Trans Queues*) to the *Dispatcher*. At runtime, the *Dispatcher* tries to obtain a batch of processed data from the *Free_Batch_Queue* and dispatches it to different GPU devices with round-robin scheduling (lines 1-11 in Algorithm 3). To reduce CPU cost, the *Dispatcher* asynchronously dispatches data (lines 9 in Algorithm 3) on a specified stream. After submitting all copying operations to GPU streams, the *Dispatcher* will be blocked to synchronize these operations submitted before (lines 13-18 in Algorithm 3), and the occupied *memory units* will be released and recycled to the *Free_Batch_Queue* for future use. In the compute engine, each GPU engine communicates with the global *Dispatcher* using a pair of *Trans Queues*. The *Trans Queue* consists of two queues that act as *Free_Batch_Queue* and *Full_Batch_Queue* in the host bridger.

4 DLBOOSTER IMPLEMENTATION

We implement our prototype for image DL workloads and integrate it with popular DL libraries to demonstrate its efficiency.

4.1 System Implementation and APIs

In hardware, we build the FPGA decoder based on OpenCL [43], where we place 4-way Huffman and 2-way resizing units according to their workloads and the constraints of FPGAs. We pack the

Algorithm 3: Asynchronously dispatching

```
   Input: solvers
   Output: None
1  foreach _solver in solvers do
2      while full_batch_queue is EMPTY do
3          full_batch_queue.blocking_wait();
4      free_trans_q ← _solver.Trans_Queues[FREE];
5      while free_trans_q is EMPTY do
6          free_trans_q.blocking_wait();
7      device_batch ← free_trans_q.pop();
8      batch_items ← full_batch_queue.pop( );
9      CudaMemcpyAsync(device_batch.decive_addr,
       batch_items.virt_addr, _solver.copy_stream);
10     working_queue[HST].push_back(hst_batch);
11     working_queue[DEV].push_back(dev_batch);
12 //sync stream to recycle memory buffers;
13 foreach _solver in solvers do
14     CudaStreamSync(_solver.copy_stream);
15     dev_bth ← working_queue[HST].pop();
16     hst_bth ← working_queue[DEV].pop();
17     _solver.Trans_Queues[FULL].push(dev_bth);
18     free_batch_queue.push(hst_bth);
19 return;
```

decoder running logic as a mirror, which can be downloaded to the FPGA devices according to different workflows.

In software, we implement *FPGAReader*, *MemManager*, and *Dispatcher* to drive the FPGA decoder to process data (Algorithm 1), manage memory (Algorithm 2), and pipeline GPU engines (Algorithm 3), respectively. Each component is connected to its neighbors by high-speed channels. These components can be scaled freely while offering simple interfaces to users. Table 1 summarizes the details of the APIs and modules.

4.2 Integrating DLBooster with DL Libraries

DLBooster can be plugged into different DL libraries flexibly and coexist with other preprocessing backends. Programmers only spend trivial effort in application modification and can obtain significant performance gains with DLBooster. We demonstrate this in Section 5.2 and 5.3: With slight modifications (200 LoC for both NVCaffe and TensorRT), both DLBooster-enabled NVCaffe and DLBooster-enabled TensorRT deliver online preprocessing services at high speed while maintaining low CPU cost.

5 EVALUATION

We evaluate DLBooster performance on two typical end-to-end DL workflows, namely, local training (Section 5.2) and online inference (Section 5.3). In the training experiment, we evaluate the DLBooster-enabled NVCaffe performance in terms of throughput and CPU cost via comparison with other backend-enabled NVCaffe frameworks. In the inference experiment, we evaluate how DLBooster-enabled NVCaffe performs in terms of throughput, latency, and CPU cost when compared to other backend-enabled TensorRT frameworks. We use a couple of DL models as the benchmarks in both experiments. More details are described as follows.

5.1 Experiment Setup

Testbed configuration: We deploy the FPGA decoder on an Intel Arria 10AX [10] FPGA device and set up our experiments on a local GPU server. The server consists of 2 NVIDIA Tesla P100s, two Intel Xeon E5-2630-v3 CPUs (32 cores in all), 64GB DRAM, a 40Gbps NIC and an Intel Optane 900p [11] NVMe disk. In the software, we run our programs on CentOS-7 with the latest NVIDIA driver (v384.145), CUDA 9.0, cuDNN v7.13, NCCL v2.2.13, and other third-party dependencies.

Compute engines: We integrate DLBooster with NVCaffe v0.17.0 [13] for local training experiment and integrate DLBooster with TensorRT v4.0 [15] for online inference. We enable LMDB and CPU-based data preprocessing backends in NVCaffe as our training baselines, and we enable nvJPEG and CPU-based data preprocessing backends in TensorRT as our inference baselines.

Models and dataset. To fully evaluate how DLBooster performs, we deploy DLBooster on several DL models in two DL workflows, i.e., LeNet-5 [37], AlexNet [35], and ResNet-18 [26] for training, and GoogLeNet [44], VGG-16 [42], and ResNet-50 [26] for inference, respectively. In training experiments, the LeNet-5 is trained on the MNIST [38], which contains 60,000 grayscale images, whereas AlexNet and ResNet-18 are trained on the ILSVRC2012 [41], which consists of more than 12.8 million color images. In the inference experiments, we simulate a network transporting scenario via a 40Gbps fabric, where 5 clients send color JPEG-formatted images (with an average size of 375×500) in real time.

5.2 Offline Training Experiment

We conduct comparative experiments to evaluate how DLBooster-enabled NVCaffe and other two baselines (i.e., CPU-based and LMDB-enabled NVCaffes) perform on Lenet-5, AlexNet, and Resnet-18, respectively, and the results are presented as follows.

Throughput: The throughput of the training experiments is shown in Figure 5. Specifically, Figure 5(a), Figure 5(b) and Figure 5(c) show the throughput when training LeNet-5, AlexNet, and ResNet-18, respectively. From the illustrated results, we prove the following: (1) DLBooster boosts NVCaffe approaching the performance boundary that GPU can achieve; (2) LMDB-enabled NVCaffe achieves high throughput during single GPU training and downgrades its throughput by 30% when training AlexNet with 2 GPUs (refer to Figure 5(b)). (3) CPU-based NVCaffe can achieve attractive throughput when training all three models.

DLBooster outperforms the two other backends and boosts the training performance of NVCaffe by 30% and 20% respectively, and the reasons can be summarized into two main aspects. (1) DLBooster exploits large-block memory to hold a batch of processed data instead of each datum, and thus eliminates the overheads introduced by copying small pieces. For example, when training LeNet-5 with NVCaffe, LMDB and CPU-based backend copy each datum to GPU in small pieces, which results in ~20% performance downgrades. (2) DLBooster implements the decoding service as a singleton and actively pipelines processed data in round-robin scheduling, thereby avoiding the imbalance among multiple threads.

Table 1: DLBooster API and module design

API	Owner	Arguments	Descriptions
`submit_cmd`	FPGAChannel	packeted cmds	Submit cmd to FPGA decoder and launch decoding operation
`drain_out`		None	Query the FPGA decoder processing signal asynchronously
`get_item`	MemManager	buffer_size	Retrieve memory from memory pool with specified size
`recycle_item`		None	Return memory buffer to memory pool for the next use
`phy2virt`		physical address	Convert physical memory address to virtual memory address
`virt2phy`		virtual address	Convert virtual memory address to physical memory address
`load_from_disk`	DataCollector	None	Obtain the metadata (blocks description) of files from disk
`load_from_net`		None	Fetch data from networking and store to the specified address

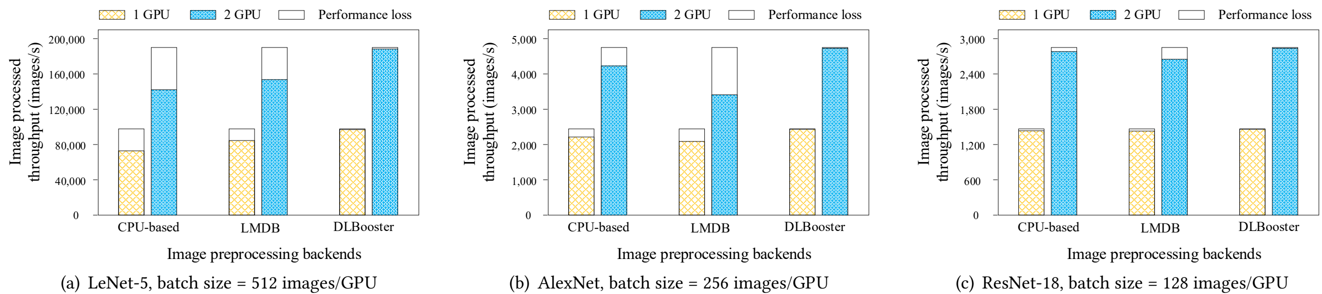


Figure 5: Training throughput for LeNet-5, AlexNet and ResNet-18 on NVCaffe with different backends. We offers the CPU resources with the best effort to achieve the maximum throughput in training with different backends.

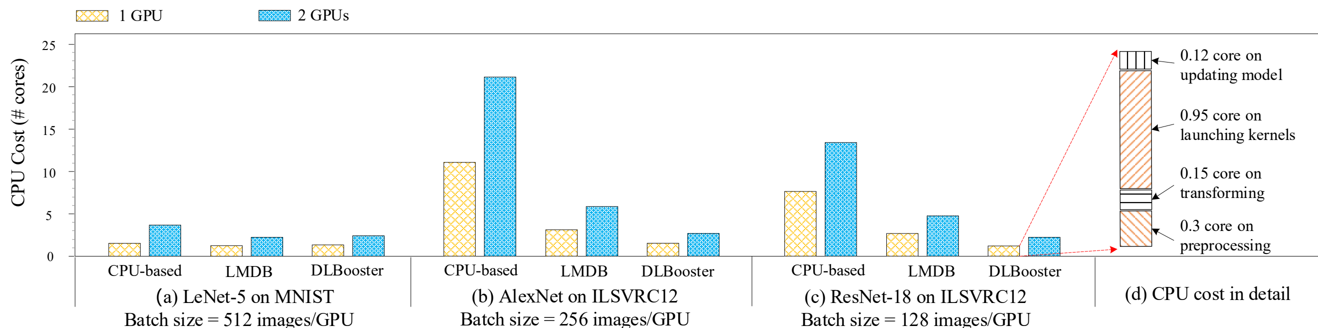


Figure 6: CPU cost in training experiments. (a)-(c) compare the CPU core cost when training LeNet-5, AlexNet, and ResNet-18 under different configurations, i.e., preprocessing backends (CPU-based, LMDB, and DLBooster) and GPU numbers; (d) gives the CPU core cost in detail when training ResNet-18 with DLBooster preprocessing backend. Training ResNet-18 with the DLBooster backend costs no more than 1.5 CPU cores in all, where preprocessing costs only 0.3 core.

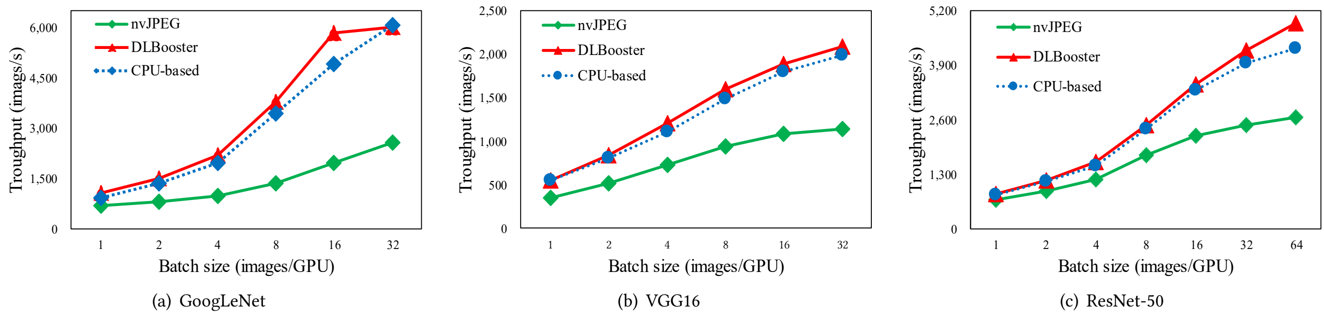
By contrast, when training AlexNet, several decoding instances will compete for shared LMDB and interact with each other, resulting in ~30% performance downgrades in LMDB-enabled NVCaffe.

CPU Overhead: The CPU consumption of training experiments is shown in Figure 6. From the illustrated results, we demonstrate the following: (1) the DLBooster-enabled NVCaffe consumes ~1.5 cores per GPU in training all three DL models; (2) the LMDB-based NVCaffe consumes ~2.5 cores per GPU when training all three DL models; (3) the CPU-based NVCaffe burns ~12 cores per GPU when training AlexNet and ~7 cores per GPU when training ResNet-18, respectively, to deliver ideal throughput for GPU engines.

Compared to the baselines, DLBooster incurs negligible CPU overheads. More specifically, we find that there is only 0.3 core occupied by data preprocessing workloads while training ResNet-18 (refer to Figure 6(d)). While training with LeNet-5, all the three backends cause little CPU overheads, because the MNIST dataset is so small that it can be cached in memory after the first epoch. However, while training with AlexNet and ResNet-18, the ILSVRC2012 dataset cannot be cached in memory and has to be loaded by CPU from disk to memory periodically, thus demonstrating the significant performance benefit of DLBooster.

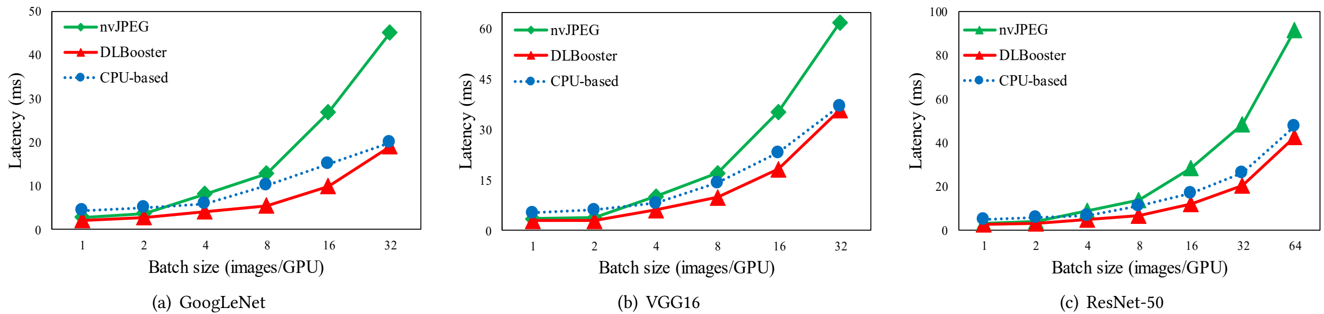
5.3 Online Inference Experiment

We also conduct comparative experiments to show how DLBooster and the baselines perform for online inference. Unlike offline training, data caching no longer provides performance benefits for online inference, because each input is used only once. Therefore, backends such as LMDB cannot boost the performance for online inference and we do not use them as baselines in the online inference experiment. Instead, we integrate DLBooster and two other online



(a) GoogLeNet (b) VGG16 (c) ResNet-50

Figure 7: Inference throughput for GoogLeNet, VGG16 and ResNet-50 on TensorRT with DLBooster backend, nvJPEG backend and CPU-base backend respectively. The default type is float16 to enable Tensor Core



(a) GoogLeNet (b) VGG16 (c) ResNet-50

Figure 8: Inference latency for GoogLeNet, VGG16 and ResNet-50 on TensorRT with DLBooster backend, nvJPEG backend and CPU-base backend, respectively. The default type is float16 to enable Tensor Core

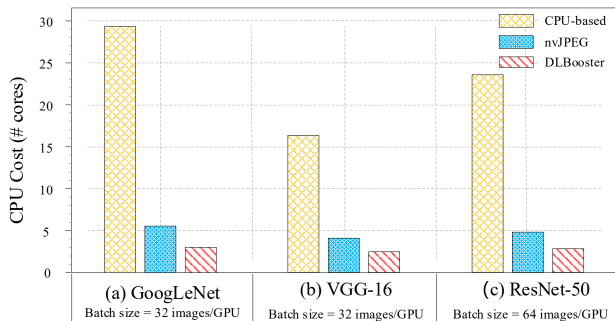


Figure 9: CPU cost in the inference experiments.

backends (i.e., CPU-based and nvJPEG) into TensorRT.[5] To simulate the online inference scenario, we set up 5 clients to send color images using a 40Gbps fabric. The average image size is 500×375, and all images are stored in JPEG format. Then, we execute the experiments with different benchmarks (i.e., GoogLeNet, VGG-16, and ResNet-50), and compare their performance on different aspects, including throughput, latency, and CPU cost.

Throughput: Figure 7 illustrates the throughput metric for different models on different backend-enabled TensorRTs, from which we obtain the following findings: (1) DLBooster-enabled TensorRT achieves 1.2×∼2.4× throughput, compared to the other two baselines. (2) nvJPEG-enabled TensorRT achieves the lowest throughput, i.e., ∼40% performance degradation as the batch size increases, because the CUDA cores are competed between the inference engine and nvJPEG. (3) The CPU-based TensorRT delivers high throughput in inferring all three models by burning many CPU cores. (4) As the batch size increases, all three backends enabled TensorRTs to deliver higher throughput as shown in Figure 7 (a)-(c).

To understand why nvJPEG-enabled TensorRT performs poorly as batch size increases, we investigate the GPU utilization and find that, in order to offer sufficient throughput to inference engines, the decoding on nvJPEG needs to consume ∼30% of GPU resources. In addition, this experiment also exposes the drawbacks of DLBooster: when the batch size is greater than 16 in Figure 7(a), DLBooster approaches its performance bound due to the drawbacks of the decoder's design. However, the bottleneck can be overcome by plugging more FPGA devices or further optimizing the decoder in FPGA. As what will be shown in Section 5.4, the cheaper price of FPGA (compared with that of CPU/GPU devices) makes it a cost-effective strategy to improve the GPU utilization by employing more FPGA decoders. Besides, we believe there is still much improvement space left with our decoder implementation. However, since this work aims to prove the effectiveness of our offloading design to boost end-to-end DL workflows, we leave the optimization of the FPGA decoder to our future work.

Latency: Latency greatly affects the user experience in online inference services. In the latency experiment, we only evaluate the time cost introduced by the online inference system and ignore the time cost due to other processing overheads. More concretely, we measure the time duration from the point when the inference system receives pictures from clients to the point when engines make a prediction as the latency metric.

[5]TensorRT is a fast GPU-based inference engine, and nvJPEG [12] is a new image preprocessing backend that migrates heavy workloads to GPU devices.

The experimental results are shown in Figure 8, from which we make the following observations: (1) DLBooster-enabled TensorRT achieves the lowest latency when compared with other two baselines, mainly because of the advantage of FPGAs in computing. (2) When the batch size is small, TensorRTs with three different backends achieve ultralow latency, i.e., 1.2 *ms*, 1.8 *ms*, and 3.4 *ms* for DLBooster, nvJPEG, and CPU-based backends, respectively. (3) As the batch size increases slightly, DLBooster and CPU-based backends vary little, whereas nvJPEG significantly increases its latency, which is mainly due to the competition for GPU cores between nvJPEG and inference engines. (4) All three TensorRTs with different backends increase their latency in different DL models, as the batch size significantly increases. The sharply increasing latency in DLBooster and CPU-based enabled TensorRTs is primarily introduced by the inference engines, while the overhead in nvJPEG-enabled TensorRT results from both competition issues and inference engines.

Besides, compared with the CPU-based data preprocessing, it is even more impractical to employ GPU for data preprocessing, because the cost of GPU resource is very expensive. According to our long-term operation experience in Azure, most of the customers cannot afford to pay such prices for just data preprocessing. Considering the statue quo of CPU-based data preprocessing, DLBooster possesses more potential advantages for online inference in large scale, because it saves the CPU resources compared to the baselines.

CPU Overhead: The CPU core consumption in the inference experiment is shown in Figure 9, and the main findings can be summarized as follows: (1) Similar to the offline training experiment, CPU-based TensorRT burns 7∼14 CPU cores per GPU in different models to deliver the online preprocessing services that the GPU wants. (2) Both nvJPEG-enabled and DLBooster-enabled TensorRTs result in little overhead for CPU core consumption (i.e., 1.5 cores per GPU in nvJPEG and 0.5 core per GPU in DLBooster), as both backends migrate preprocessing workloads to other devices (i.e., FPGA in DLBooster and GPU in nvJPEG). However, in the nvJPEG backend, few (1∼2) CPU cores are used to launch CUDA kernels. Moreover, the nvJPEG-enabled frameworks burn GPU cores and downgrade the performance of the model computing part.

5.4 Economic Analysis

From the economic perspective, we can also prove that DLBooster possesses great benefits for online data preprocessing services:

(1) **To common users:** DLBooster provides the following economic gains to common users. First, it offers online data preprocessing services and avoids the time-consuming data conversion in offline backends. Second, DLBooster offers high-performance services by offloading workloads to FPGAs instead of burning CPU/GPU cores. Both advantages can save more money for users to migrate DL workloads to the cloud.

(2) **To cloud providers:** Today, Azure and other cloud providers sell cloud services such as VMs, including GPU instances, to the public. According to Azure [19], a physical core (2 hyperthreads) on the cloud sells for $0.10∼0.11 per hour, or potential revenue of ∼$900 per year. In modern DL workflows, a well-optimized FPGA decoder [9] can offer the same online data preprocessing services as 30 cores. By enabling DLBooster, those 30 cores can be replaced by one FPGA, and the saved CPU cores can still be sold to other tenants for more than $1.5/h. Moreover, FPGAs have the lowest power consumption (∼25W [20]) compared with CPU (∼ 130W) and GPU (∼ 250W), and thus reduce the cost of cloud maintenance.

Summary: Based on the experiments and analysis in Section 5.2∼ Section 5.4, we demonstrated that DLBooster can achieve great performance gains (including throughput, latency, CPU overhead, etc.) compared to existing data preprocessing backends. Meanwhile, it is also a cost-effective solution that brings much economic benefit to both cloud service customers and providers.

6 RELATED WORK

In recent years, DL acceleration has been extensively studied due to the increasing data volume and growing model complexity. However, most of them focus on the optimization of computing and communication, but rarely consider the impact of data preprocessing on end-to-end DL workflows. Existing works on the data preprocessing of DL systems can be organized as follows:

Offline and online data preprocessing: The mainstream DL libraries usually have their data preprocessing backends. In general, these backends can be divided into two categories based on their service primitives. The first category is composed of offline primitives, such as *RecordIO* [2] in MXNet, *LMDB* [50] in Caffe, and *TFRecord* [17] in TensorFlow. This offline service usually requires significant efforts to convert the original data to the data with an identifiable format before "learning". The second category is composed of online primitives, which achieve high-performance services by burning many CPU cores and therefore are not scalable. DALI [12], proposed by NVIDIA, is a hybrid data preprocessing backend that exploits CPU cores and GPU-based nvJPEG [16] to process decoding workloads. However, when there are no enough CUDA cores to use, the GPU-based nvJPEG may downgrade the overall performance of the end-to-end DL workflow by competing for GPU resources.

Pipelining input with fast IO: As more powerful and larger-scale GPU clusters are available today, DL workflow has been dramatically speeded up [24, 27, 30, 47], putting a heavier demand on the input. Pioneering studies [24, 30] have noted the mismatch between the straggling input modules and faster GPU engines, and they try to mitigate this by leveraging faster IO and better pipelining. For example, [24] places 8 NVMe SSDs into each server to achieve fast IO in their experiments. [30] pipelines the input module of TensorFlow to minimize both CPU and GPU idle time. However, naively pipelining data input on fast IO can hardly alleviate the massive gap between supply and demand for data preprocessing in the DL system. Unlike the existing solutions, DLBooster offloads the preprocessing logic to FPGA hardware and elaborately designs the processing logic, and it thus can eliminate the bottleneck and achieve much higher performance for various DL workflows.

7 CONCLUSIONS AND FUTURE WORK

Data preprocessing, which is mostly ignored by existing DL acceleration solutions, is becoming the bottleneck in end-to-end DL workflows. In order to accelerate the DL workflow and avoid the preprocessing bottleneck, we propose DLBooster, to boost the overall

performance and release the heavy burdens on CPUs. Our experiments on both offline training and online inference demonstrate that, DLBooster can achieve 1.3×∼2.4× overall performance and reduce latency by 1/3 in typical DL benchmarks when compared with other data preprocessing backends. As for the future improvement with DLBooster, there are several directions in consideration, including (1) tuning the decoder in FPGA to achieve higher utilization and throughput, (2) directly writing the processed data to GPU devices for lower latency, (3) extending more preprocessing kernels for more DL applications.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant 2018YFB1800800, 2018YFB-1800600, the Research and Development Program in Key Areas of Guangdong Province (Grant No.2018B010113001), and the National Natural Science Foundation of China under Grant No. 61432002, No. 61772305, No.61672499. Dan Li is the corresponding author of this paper.

REFERENCES

[1] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.
[2] Anirudh Acharya et al. 2018. Image Transforms and RecordIO file Creation of MXNet. https://cwiki.apache.org/confluence/display/MXNET/Image+Transforms+and+RecordIO+file+Creation.
[3] Takuya Akiba et al. 2017. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325* (2017).
[4] Léon Bottou et al. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.
[5] Jianmin Chen et al. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).
[6] Tianqi Chen et al. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274* (2015).
[7] Yang Cheng et al. 2019. Bridging machine learning and computer network research: a survey. *CCF Transactions on Networking* 1, 1 (May 2019), 1–15.
[8] Minsik Cho et al. 2017. PowerAI DDL. *arXiv preprint arXiv:1708.02188* (2017).
[9] Intel Corporation. 2018. A brief JPEG decoder example design on Intel Stritax V A7 FPGA. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/support/examples/download/exm_opencl_jpegdecoder.pdf.
[10] Intel Corporation. 2018. Intel Arria-10 FPGAs. https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.
[11] Intel Corporation. 2019. A brief Introduction to the Intel Optane SSD 900p Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series.html.
[12] NVIDIA Corporation. 2018. Announcing NVIDIA DALI and NVIDIA nvJPEG. https://news.developer.nvidia.com/announcing-nvidia-dali-and-nvidia-nvjpeg/.
[13] NVIDIA Corporation. 2018. NVCaffe, an NVIDIA-maintained fork of Berkeley Vision and Learning Center (BVLC) Caffe tuned for NVIDIA GPUs. https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/caffe/.
[14] NVIDIA Corporation. 2018. NVIDIA DGX-2: the world's most powerful AI system for the most complex AI challenges. https://www.nvidia.com/en-us/data-center/dgx-2/.
[15] NVIDIA Corporation. 2018. NVIDIA TensorRT Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.
[16] NVIDIA Corporation. 2018. nvJPEG GPU-accelerated JPEG decoder. https://developer.nvidia.com/nvjpeg.
[17] Mark Daoust et al. 2018. Using TFRecords and tf.Example in TensorFlow. https://www.tensorflow.org/tutorials/load_data/tf-records.
[18] Ltsc Deng et al. 2013. Recent advances in deep learning for speech research at Microsoft.. In *ICASSP*, Vol. 26. 64.
[19] Daniel Firestone et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of NSDI'18, Renton, WA*.
[20] Robin Flowerdew et al. 1991. Using areal interpolation methods in geographic information systems. *Papers in regional science* 70, 3 (1991), 303–315.
[21] Jort F. Gemmeke et al. 2017. Audio Set: An ontology and human-labeled dataset for audio events. In *Proc. IEEE ICASSP 2017*. New Orleans, LA.
[22] Jinkun Geng et al. 2018. HiPS: Hierarchical Parameter Synchronization in Large-Scale Distributed Machine Learning. In *Proceedings of NetAI'18*.
[23] Jinkun Geng et al. 2019. Rima:An RDMA-Accelerated Model-Parallelized Solution to Large-Scale Matrix Factorization. In *ICDE'19*.
[24] Priya Goyal et al. 2017. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
[25] Chuanxiong Guo et al. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of ACM SIGCOMM '16*.
[26] Kaiming He et al. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.
[27] Kaiming He et al. 2016. Deep residual learning for image recognition. In *IEEE CVPR'18*. 770–778.
[28] Yukai Huang et al. 2017. LOS: A High Performance and Compatible User-level Network Operating System. In *APNet'17*.
[29] M. Jaderberg et al. 2014. Reading Text in the Wild with Convolutional Neural Networks. *arXiv preprint arXiv:1412.1842* (2014).
[30] Xianyan Jia et al. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205* (2018).
[31] Yangqing Jia et al. 2014. Caffe: Convolutional architecture for fast feature embedding. In *the 22nd ACM Multimedia*. ACM, 675–678.
[32] Geng Jinkun et al. 2019. ElasticPipe: An Efficient and Dynamic Model-Parallel Solution to DNN Training. In *ScienceCloud '19*.
[33] Geng Jinkun et al. 2019. Horizontal or Vertical? A Hybrid Approach to Large-Scale Distributed Machine Learning. In *CCIW '19*.
[34] Nikhil Ketkar. 2017. Introduction to pytorch. In *Deep Learning with Python*. Springer, 195–208.
[35] Alex Krizhevsky et al. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS 2012*. 1097–1105.
[36] Quoc V Le. 2013. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 8595–8598.
[37] Yann LeCun et al. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, Vol. 60. Perth, Australia, 53–60.
[38] Yann LeCun et al. 1998. THE MNIST DATABASE of handwritten digits. Retrieved 2012 from http://yann.lecun.com/exdb/mnist/
[39] Yann LeCun et al. 2015. Deep learning. *nature* 521, 7553 (2015), 436.
[40] Y. Li et al. 2018. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *MICRO'18*.
[41] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252.
[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[43] John E Stone et al. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.
[44] Christian Szegedy et al. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.
[45] Trinh Xuan Tuan et al. 2017. 3D Convolutional Networks for Session-based Recommendation with Content Features. In *Proceedings of RecSys '17*.
[46] Shuai Wang et al. 2019. Impact of Network Topology on the Performance of DML: Theoretical Analysis and Practical Factors. In *IEEE INFOCOM 2019*.
[47] Yang You et al. 2017. ImageNet training in minutes. *CoRR, abs/1709.05011* (2017).
[48] Yibo Zhu et al. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of ACM SIGCOMM '15*.
[49] Martin Zinkevich et al. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*. 2595–2603.
[50] Feng Zou. 2017. How to create ImageNet LMDB in Caffe. https://github.com/intel/caffe/wiki/How-to-create-Imagenet-LMDB.