Accelerating End-to-End Deep Learning Workflow With Codesign of Data Preprocessing and Scheduling

Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jinkun Geng, Wei Bai, Jianping Wu, and Yongqiang Xiong

Abstract—In this article, we investigate the performance bottleneck of existing deep learning (DL) systems and propose DLBooster to improve the running efficiency of deploying DL applications on GPU clusters. At its core, DLBooster leverages two-level optimizations to boost the end-to-end DL workflow. On the one hand, DLBooster selectively offloads some key decoding workloads to FPGAs to provide high-performance online data preprocessing services to the computing engine. On the other hand, DLBooster reorganizes the computational workloads of training neural networks with the backpropagation algorithm and schedules them according to their dependencies to improve the utilization of GPUs at runtime. Based on our experiments, we demonstrate that compared with baselines, DLBooster can improve the image processing throughput by $1.4\times - 2.5\times$ and reduce the processing latency by 1/3 in several real-world DL applications and datasets. Moreover, DLBooster consumes less than 1 CPU core to manage FPGA devices at runtime, which is at least 90 percent less than the baselines in some cases. DLBooster shows its potential to accelerate DL workflows in the cloud.

Index Terms—Deep learning, data preprocessing, workload offloading, computation scheduling, FPGAs

1 Introduction

Nowadays deep learning (DL) has led to great success in many areas, such as computer vision (CV) [2], natural language processing (NLP) [3], etc. Given the prevalence of DL, we have witnessed many efforts made to accelerate DL workloads during the past few years, especially in the cloud, where more than 96 percent of DL applications [4] are deployed today. For instance, many DL frameworks, such as MXNet [5], PyTorch [6], and TensorFlow [7], are well optimized for the cloud, serving as cloud infrastructure toolkits. High-performance computation and communication technologies are also rapidly developing [8] to speed up the compute- and communication-intensive DL workloads. For example, emerging hardware accelerators, such as GPUs [9], TPUs [10], and FPGAs [11], have been widely used to speed up DL training/inference. Remote Direct Memory Access (RDMA) [12], which achieves high throughput and low latency with near-zero CPU overhead, is used to improve the communication of distributed DL workloads [13].

Having revisited a massive volume of prior works [14], [15], [16], [17], we find that most of them focus on how to speed up the workloads of training/inferring complex neural networks (NNs) [3], [18], [19], but ignore the other parts of the end-to-end DL workflow, such as data preprocessing, in which raw data with various formats and shapes are converted into the unified feature vectors to be fed into NNs for training/inference. As a fundamental step of DL workflows, data preprocessing can put a significant impact on the overall performance of DL applications. Moreover, as more emerging hardware and software optimizations are available, the training/inference speed has been greatly improved [14], [16], making the data preprocessing an increasing bottleneck in DL workflows. Particularly in the experiment of training AlexNet [2] on a GPU cluster, we find that: (1) When using LMDB [20] as the data preprocessing backend, Caffe [21] loses training performance (i.e., image processing throughput) by up to 40 percent. (2) After switching to the runtime data preprocessing with 4 CPU cores (by default), the throughput even degrades by up to 60 percent, and the training performance gaps of Caffe are finally made up by burning as many as 12 CPU cores for each GPU.

As a compute-intensive application, efficient data preprocessing backends are urgently needed by deploying DL workflows on GPUs. Revisiting the data preprocessing backends widely used today, we find that they often fail to provide satisfying data preprocessing services with several limitations, particularly in the cloud. For example, the offline data preprocessing backends, such as TFRecord [22],

- *Yang Cheng, Dan Li, Jinkun Geng, and Jianping Wu are with Tsinghua University, Beijing 100084, China. E-mail: cheng-y16@mails.tsinghua.edu.cn, tolidan@tsinghua.edu.cn, steam1994@163.com, jianping@cernet.edu.cn.*
- *Zhiyuan Guo is with Beihang University, Beijing 100191, China and also with Microsoft Research, Beijing 100080, China. E-mail: v-zhguo@microsoft.com.*
- *Binyao Jiang is with Shanghai Jiao Tong University, Shanghai 200240, China , and also with Microsoft Research, Beijing 100080, China. E-mail: v-bijian@microsoft.com.*
- *Wei Bai and Yongqiang Xiong are with Microsoft Research, Beijing 100080, China. E-mail: {weibai, yongqiang-xiong}@microsoft.com.*

Manuscript received 1 July 2020; revised 24 Oct. 2020; accepted 30 Nov. 2020. Date of publication 29 Dec. 2020; date of current version 11 Feb. 2021. (Corresponding author: Dan Li.) Recommended for acceptance by P. Balaji, J. Zhai, and M. Si. Digital Object Identifier no. 10.1109/TPDS.2020.3047966

RecordIO [23], and LMDB [20], introduce significant time costs when processing the training data offline first, thus not suitable for the online inference tasks. Burning a volume of CPU/GPU cores [24], [25] at runtime is another choice to offer online data preprocessing services to the NNs training/inferring. However, it also has the following shortcomings: (1) First, burning too many CPU/GPU cores for data processing will degrade the training/inference performance by competing for the CPU/GPU cores with them. (2) Second, including the offline backends, those approaches introduce expensive costs to both cloud users and cloud service providers: (i) On the one hand, cloud users are expensively charged for the resources they used in terms of both time and amount. (ii) On the other hand, burning too many CPU/GPU cores can result in high power consumption, thus increasing the maintenance costs of the cloud. To name but a few, there are many defects (Section 2.2) widely existing in current DL systems [20], [22], [23], [24], [25] but not well solved, making data preprocessing the distinct bottleneck of DL workflows, especially in the cloud.

Benefiting from the programming flexibility while providing hardware-like performance [26], the field-programmable gate array (FPGA) [11] enables us to offload heavy workloads on it to reduce the burden on software systems. After several years of rapid development, FPGAs have built a maturing ecosystem [27], and the application of FPGAs has led to a revolution in many areas [28], [29]. Meanwhile, the declining price and low power consumption of FPGA devices (Section 7) also strengthen the advantages of running large-scale FPGA-based applications. Today, FPGAs have been deployed [29], [30], [31] at scale to speed up emerging workloads in the cloud, such as NNs training [28], networking service [26], etc. The success of FPGAs in speeding up cutting-edge applications also shows its potential to improve the performance of DL tasks.

Considering the distinct performance bottleneck of data preprocessing in DL workflows and the potential of FPGAs to accelerate cutting-edge applications, we propose DLBooster in this work to meet the increasing demands on data preprocessing from emerging DL applications. DLBooster provides online data preprocessing services with high performance by offloading some key decoding workloads to FPGAs. With DLBooster, DL systems can efficiently process data and move them into GPU engines with high speed when training/inferring NNs in a multi-GPU cluster. However, DLBooster is still faced with the following challenges:

(1) *Codesign of Hardware and Software*. Different from software systems, DLBooster is a system which needs to consider the coexistence of software and hardware. For example, (i) how to deal with the different data accessing approaches between FPGA hardware and userspace programs? (ii) How to design the communication channels and primitives to pipeline the decoding modules for high performance? We will discuss them in Sections 3, 4.2.1 and 4.2.2.

(2) *Balance Between Resource Constraint and Workload*. Naively offloading all data preprocessing workloads to FPGAs is quite inefficient, limited by the hardware constraints. Therefore, we need to rethink what kind of workloads to offload and how to organize them in FPGAs to achieve the expected performance. Those considerations and design details will be well discussed in Sections 3.1 and 4.1.

(3) *Compatibility With Different DL Applications and Settings*. Serving as a general-propose data preprocessing backend, DLBooster should keep open to the mainstream DL applications and frameworks. To this end, we design DLBooster with the pluggable FPGA decoder (Section 4.1) and expose simple yet efficient APIs, so that it can be easily integrated into popular DL frameworks for use in practice.

With careful designs, we implement the prototype of DLBooster and integrate it into two popular DL engines (i.e., TensorRT [32] and NVCaffe [33]) with slight modifications. Moreover, we demonstrate the efficiency of DLBooster with experiments on two real-world image processing tasks, namely, local training and online inference. We find that DLBooster can not only improve the image processing throughput by 1.4×–2.5×, but also reduce 1/3 image processing time. Moreover, DLBooster consumes less than 1 CPU core, which is at least 90 percent less than the baselines in some cases.

In addition to the data preprocessing optimization, we have also focused on improving the computation efficiency when training NNs on GPUs via the backpropagation (BP) algorithm (Section 5). Given that modern NNs usually contain many layers, which involve different computational workloads (varied CUDA kernels [34]) with dependencies, efficiently training NNs requires fully utilizing the hardware resources when executing the kernels on GPUs. However, the overhead of launching kernels is non-negligible [35], [36], particularly for small layers where the computing resources are not fully utilized. To mitigate the impact, we separate the error propagating computations from the gradients computing workloads according to their dependencies and dispatch them onto multiple CUDA streams with different priorities [37]. In this way, computations for the two parts (in different layers) are overlapped, and the GPU utilization is also improved. Our experiment on NVCaffe shows that the training performance can be further improved by up to 14 percent by scheduling the BP workflow (Section 6.1.2).

In summary, the main contributions of this paper are:

- We reveal the increasing distinct bottleneck of data preprocessing in current DL workflows with comprehensive studies and analyses.
- We design an online data preprocessing backend by selectively offloading key workloads to FPGAs to provide the decoding service in DL workflows.
- We reorganize and schedule the workloads in the BP workflow according to their dependencies to improve the hardware utilization when training NNs on the GPU.
- We demonstrate the potential of DLBooster to accelerate emerging DL workflows with experiments on several real-world DL applications and datasets.

The rest of this paper is organized as follows: Section 2 presents the background of the end-to-end DL workflow, showing the challenges of modern DL deployments. Sections 3, 4, and 5 present the design details of DLBooster. Section 6 demonstrates the efficiency of DLBooster with experiments on typical real-world image processing tasks

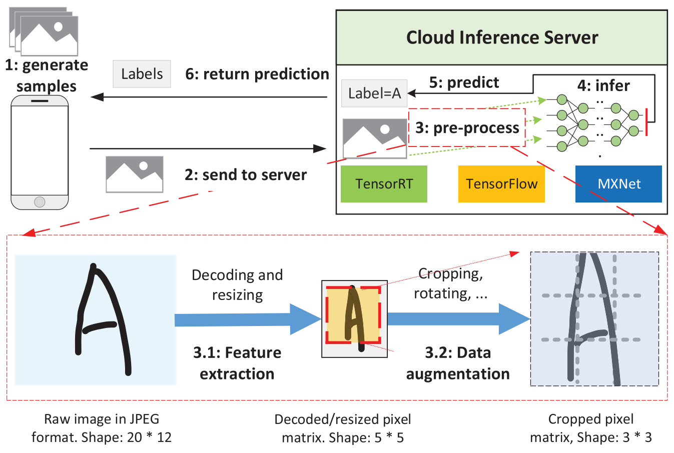


Fig. 1. An example of the online image inference task to present the end-to-end DL workflow.

and datasets. We discuss the benefits and optimizations of DLBooster in Section 7, present the related work in Section 8, and conclude this work in Section 9.

2 Background and Motivation

2.1 End-to-End DL Workflow

DL [38] is an efficient approach that uses a large number of connected neurons to solve complex optimization tasks today. Fig. 1 presents the details of the DL workflow from an end-to-end view, which consists of the following parts.

Data collecting is the first step in the DL workflow. Nowadays training complex DL models relies on rich datasets. In general, data collecting is task-specific and costly, which requires great efforts of users to collect massive volumes of data samples according to their tasks. To facilitate the application of DL, there are many open free task-specific datasets maintained by the communities, such as ImageNet [39], AudioSet [40], MJSynth [41], etc.

Data preprocessing converts the raw data in various formats into the unified input of NNs, and it can be divided into two steps in general. The first is feature extraction, where data features are extracted and reshaped to match the input layer of NNs. The second is data augmentation, which improves the diversity of datasets to avoid the *overfitting* problem when training NNs with small datasets. As a fundamental step in the DL workflow, data preprocessing tackles volumes of data in various formats for different DL tasks. For instance, in image processing tasks [2], the pixel matrices are extracted from the raw images in various formats (e.g., PNG, JPEG). In speech processing tasks [42], the spectra information of audio is decoded by the discrete cosine transform (DCT). In NLP tasks [3], the vectorized features are extracted from the texts in different languages.

Training/inference is the core part of DL applications. (1) When DL models (i.e., NNs) are first built, they will be trained with datasets to model the tasks. The stochastic gradient descent (SGD) [43] is the de facto algorithm of training NNs with iterative computation. (2) Being well trained, the DL models will be inferred to make predictions to the unknown data with high accuracy. Different to training, DL models are only forwarded once in the inference workflow.

TABLE 1
Profiling of Using Caffe[21] to Train AlexNet With Real-World Dataset on a GPU Cluster

Metric	x 1 P100 GPU (2512 images/s)			x 2 P100 GPUs (4648 images/s)		
	LMDB	CPU (default)	CPU (max perf.)	LMDB	CPU (default)	CPU (max perf.)
Throughput (images/s)	2320	987	2280	3230	1820	4318
CPU costs (# of cores)	3	4	10	5	8	21
GPU training perf. loss	8%	60.4%	9.2%	39.1%	60.8%	7.1%

The batch size is 256 images/GPU.

Today, NNs usually go deeper (i.e., DNNs) [18], and training such DNNs is costly. Many accelerating technologies (e.g., GPU/TPU/FPGA accelerators and distributed training [43]) are studied to speed up the DL workloads. As DL applications are becoming popular in the cloud [4], where rich computation resources are available, it is increasingly important to improve the efficiency of DL workflows.

2.2 Defects in Existing Data Preprocessing Backends

Today, many DL frameworks have provided the data preprocessing backends to process huge volumes of data, which can be categorized into two types, namely, *online primitive* and *offline primitive*. The *offline primitive* processes the training data in advance and loads the processed data from lightweight databases (DBs) for next time use, such as TFRecord [22] in TensorFlow, LMDB [20] in Caffe, RecordIO [23] in MXNet, etc. On the contrary, the *online primitive* provides high-performance online data preprocessing services by burning a lot of CPUs/GPUs to decode at runtime.

However, according to our studies, most data preprocessing backends widely used today are inefficient with several limitations. Particularly, in our experiment (summarized as Table 1) on training AlexNet with the ILSVRC12 [39] dataset in a GPU cluster (2 P100 GPUs), we observe 39.1 and 60.8 percent of training performance loss in Caffe [21] when using LMDB and the CPU-based backend (with default configurations) respectively, compared to training with synthetic data. Caffe using the CPU-based backend makes up training performance gaps by burning around 10 CPU cores for each GPU. Those data preprocessing backends are becoming the bottleneck in modern DL workflows, which can be more severe when GPUs are faster. The defects of existing data preprocessing designs are described as follows.

(1) *Performance Degradation.* Both online and offline data preprocessing backends suffer from performance degradation for the following reasons:

- The *offline primitive* backends such as LMDB require a fine-grained data cache mechanism and a large memory configuration, which is not always met when datasets grow larger. For instance, in our experiment, the LMDB-enabled Caffe loses the training performance because LMDB runs out of memory and reloads data in large blocks (the processed data are much larger) from disks.

- The *offline primitive* backends are inefficient or cannot work in some DL tasks such as online inference, which requires ultralow processing latency.
- The *online primitive* backends, such as nvJPEG [24] and the CPU-based backend, are blamed for burning too many GPU/CPU cores at runtime, which results in they competing for the limited computing resources with the other modules in an end-to-end DL workflow, such as parameter aggregation [44] in distributed training, or the computation of inferring/training NNs [32].
- The *online primitive* backends scale poorly as more GPUs are inserted into a server. For example, in DGX-2 [9], which contains 2 Xeon E5 CPUs (48 cores in all) and 16 V100 GPUs, each V100 GPU can use only 3 CPU cores on average. However, we find that a V100 GPU can process as many as 5,000 images per second in the inference of ResNet-50, while each CPU core can only provide online data preprocessing services of approximately 300 images per second. There is a big mismatch between GPU/CPU performance and configurations.

(2) *Expensive Cost*. Currently, many data preprocessing designs that are widely used today are suffering from expensive costs in terms of both economy and time:

- The *offline primitive* backends usually take a lot of time to process the data in advance. For instance, it takes us 2 hours to convert ILSVRC12 [39] into the LMDB-formatted data.[1] The nontrivial time costs harm the interests of users who deploy DL tasks on the cloud, because VM instances are sold at an expensive price based on the time they are occupied.
- The *online primitive* backends usually burn a lot of CPU/GPU cores to process the data at runtime, which bring expensive costs to both cloud providers and users for the following reasons: (i) Both GPUs and CPUs are much expensive in the cloud. (ii) The high power consumption increases the maintenance costs of the cloud.

2.3 Computation Efficiency of Training NNs on GPU

Currently, many high-performance computing accelerators are used to speed up the training of modern NNs in practice, such as GPUs. Different to the programs running on CPUs, GPU-based programs run in an asynchronous way for the parallel computing performance of GPU hardware. Therefore, a fine-grained computation managing mechanism is desired to schedule a large number of computing tasks running in parallel to make full use of the GPU computing resources, thus boosting the performance of applications.

Training NNs is a typical application of GPUs, and many advanced software libraries provide well-optimized DL operators to boost the computation for NNs on GPUs, such as cuBLAS [34] and cuDNN [45]. In fact, neurons are usually organized layer by layer in modern NNs [18], [19], and there are dependencies between these layers. Generally, each layer has different computational workloads. However, we note that (1) limited by the dependency, computational workloads at different layers cannot be executed in parallel. (2) It is nontrivial to launch a CUDA kernel on the GPU [46], especially for the small computing event. The characteristics would make it hard to promise that GPU computing resources are fully utilized at any time [46] during the training, even with the fine-grained DL operators which lack the global view of the entire DL workflow. Particularly, we have observed that the GPU utilization is only around 90 percent when training ResNet-50 with a large batch size, and it is even lower in the training with small batch size settings.

1. Recent works [14], [15], [16], [17] show that training ResNet-50 can be done in minutes, but we find that they do not take the data preprocessing into considerations, which is impractical.

2.4 Our Motivation

From the above studies, we conclude that: (1) The widely used data preprocessing backends suffer from several defects, making them the increasing bottleneck in modern DL workflows. Given the rapid development of GPUs, this situation will get worse. (2) The computation efficiency of training NNs on GPUs is closely related to the hardware utilization (varied workloads), which can be further improved by the fine-grained scheduling. These observations motivate us to propose DLBooster with the codesign of data preprocessing (Section 4) and computational workloads scheduling (Section 5) to speed up emerging DL workflows in the cloud.

3 DLBooster Design: Overview

3.1 Design Principles

We design DLBooster with the following considerations:

Offering Hybrid Data Preprocessing Service. Having witnessed the defects of the offline backends (Section 2.2), DLBooster is designed as an online backend. However, DLBooster can also cache the processed data to avoid repeated workloads in iterative DL tasks (e.g., training), in which DLBooster processes all raw data in the first epoch and caches the processed data on memory/disks for future reuse.

Selectively Offloading Decoding Workloads to FPGAs. Different from other online data preprocessing backends, which burn a volume of CPU/GPU cores, DLBooster offloads the decoding workloads to FPGAs with the following considerations: (1) CPUs and GPUs are irreplaceably required by other workloads in the DL workflow, such as parameter aggregating in distributed training or computing NNs for training/inference. (2) FPGAs can achieve the competitive decoding performance at a lower price.

However, to achieve the balance between the computational workloads and hardware constraints, we choose to offload partial decoding workloads that can be efficiently executed on hardware to FPGAs, instead of all the workloads. For example, in the design of the image processing decoder demo, we only offload the Huffman decoding and resizing workloads to FPGAs for fast data processing and leave the data augmentation to GPUs to make full of the large volume of GPU memory.

Keeping Programming Flexibility. A big concern about deploying FPGAs at scale is the flexibility of programming

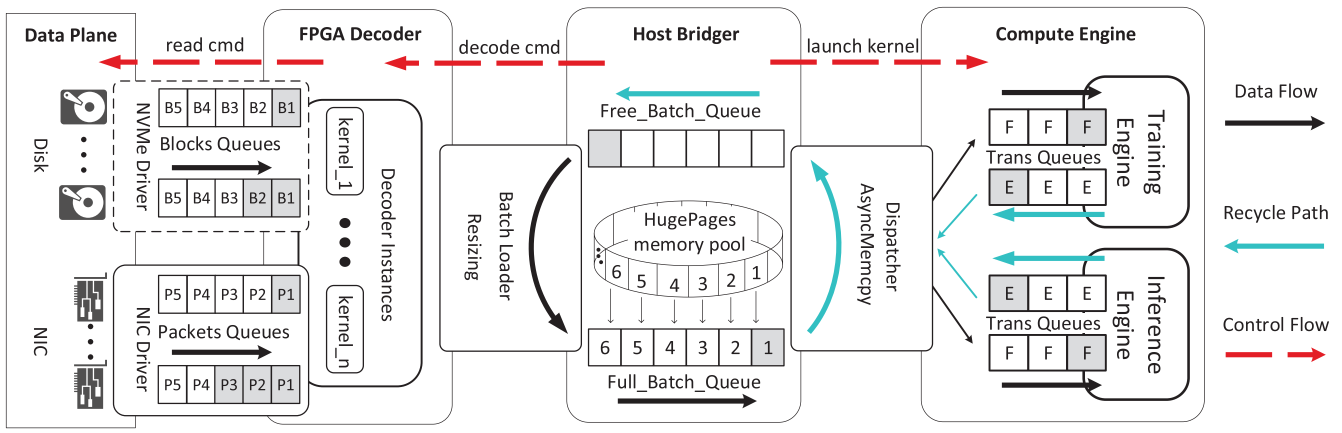


Fig. 2. DLBooster architecture. From the logical view, there are 4 layers used to organize and accommodate multiply devices. Each layer is connected with its neighboring layers. Data are moved in the directions: NIC/disk → FPGA decoder → host bridger → GPU device.

with FPGA hardware. To this end, we have the following considerations to design DLBooster as a general data preprocessing backend: (1) In the design of the FPGA decoder, DLBooster allows users to rebuild various decoding logic with OpenCL [47] and freely download the corresponding decoding logic to the FPGA device for different DL tasks (e.g., speech/image/video). (2) As for the flexibility of use, DLBooster coexists with other data preprocessing backends, benefiting from the isolation design. Moreover, DLBooster provides simple interfaces to users, which allow them to integrate it into different DL systems with less effort.

3.2 DLBooster Architecture

DLBooster codesigns hardware and software, and the architecture of DLBooster is shown in Fig. 2. From the logical view, DLBooster is composed of four layers, namely, data plane, FPGA decoder, host bridger, and computing engine, which we present in a bottom-to-top order as follows.

The data plane is the first layer of DLBooster. It caches data on local disks or fetches them from the Internet (NIC) for different DL tasks. The fetched data are then fed into the decoder in FPGAs for future use.

The FPGA decoder is located in the FPGA device, and it is used to process the data at runtime. It fetches the raw data from the data plane and processes them. After that, it delivers the processed data to the host bridger. The decoding logic is task-specific and replaceable, which allows users to rebuild the desired decoding logic for various tasks.

The host bridger is the key part of DLBooster, which connects the decoder in FPGAs with the GPU computing engines. The host bridger contains three sub-modules, namely, *FPGAReader*, *GPUHandler*, and *Dispatcher*. The *FPGAReader* is an abstraction of the decoder in FPGAs to manage it in user space, whereas the *GPUHandler* offers operating interfaces to manage the computing engine instances in GPUs. The *Dispatcher* is built to move the data from the host memory to the GPU devices in computing engines with a memory pool, after they are processed by the FPGA decoder.

The computing engine, which is responsible for the training/inference workloads in GPUs, is the top layer of DLBooster. Each GPU is invisible to the others for isolation and accesses the processed data from a high-speed channel (i.e., *Trans Queue*) controlled by the *Dispatcher*.

All the adjacent components are connected with each other by high-speed channels to provide high-performance online data preprocessing services.

4 Data Preprocessing Design

4.1 FPGA Decoder Design

DLBooster builds a decoder with different decoding logic in FPGAs to handle various data preprocessing workloads, such as image/video processing, NLP, etc. The decoder in FPGAs is controlled by the *FPGAReader* in the host bridger through two control queues, namely, *task queue* and *completion queue*. We use the example of how a raw image is processed and delivered in the image processing task to present the decoder design in detail (shown in Fig. 3).

At first, a decoding *cmd* is sent to the FPGA decoder via the *task queue*. The *cmd* consists of two entries, namely, the metainfo (i.e., data indexing, format, location, and shape) of the raw image and a physical address of host memory to receive the processed data. Then, the *Data loader* extracts the two entries with *CMD parser*, records the host memory address by *MMU* and retrieves the raw image with *DataReader* according to the metainfo. After that, the fetched raw image is sent to the decoding kernel, in which it is processed by the *Huffman decoding* unit and the *iDCT* unit in orders to extract and recover the pixel data. The extracted pixel matrix is then reshaped by the *resizing* unit and written to the host memory (recorded in *MMU*) via DMA. Finally, the *Finish arbiter* sends a *finish signal* to the corresponding *FPGAReader* in the host bridger via the *completion queue*,

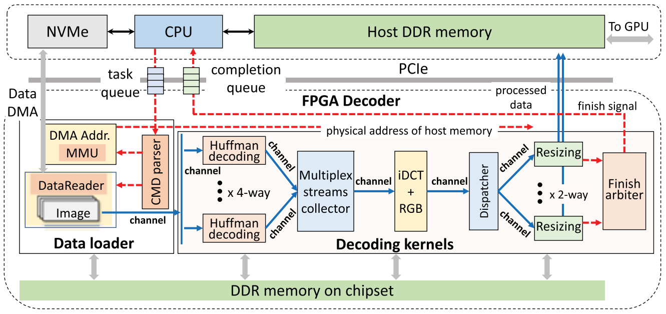


Fig. 3. FPGA decoder architecture, using the image processing task as an example to show the details.

indicating that the raw image has been correctly processed and delivered.

We further optimize the FPGA decoder as follows: (1) We divide the decoding logic into three module units (i.e., *Huffman decoding*, *iDCT* and *resizing*) and scale each of them with different number of instances according to their workloads. For instance, there are a 2-way *resizing* unit and a 4-way *Huffman decoding* unit instantiated in our demo (Section 6). (2) All units run asynchronously in the decoding pipeline to process the data in parallel. (3) Data are processed in batches. All data in one batch share one decoding *cmd* to avoid the overhead of frequent communication between the FPGA decoder and the host bridger.

As described in Algorithm 1: At first, *FPGAReader* initializes *DataCollector*, *FPGAChannel* and *mem_pool* to prepare for the following decoding services. In `DecodingLoop`, the *FPGAReader* tries to get a free memory block from the *mem_pool* or recycles from the *FPGAChannel* (lines 9–14). After that, it encodes a *cmd* with a batch of data metainfo and a physical address of host memory (lines 15–18) and submits the *cmd* to the decoder in FPGAs via the *FPGAChannel* (line 19).

The *FPGAReader* runs in a totally asynchronous way, in which it can simultaneously submit a mass of decoding *cmds* without blocking. Therefore, *FPGAReader* can achieve high decoding throughput while keeping low latency.

Algorithm 1. Asynchronous FPGAReader

```
1: Initilization ($data\_list$, $mem\_pool$, $dev\_ID$)
2:   $data\_collector \leftarrow$ getMetaInfo ($data\_list$)
3:   $fpga\_channel \leftarrow$ FPGAInit ($dev\_ID$)
4:   $free\_batch\_q \leftarrow mem\_pool.free\_batch\_queue$
5:   $full\_batch\_q \leftarrow mem\_pool.full\_batch\_queue$
6:   return
7: DecodingLoop ()
8:   while Running do
9:     $mem\_hoder \leftarrow free\_batch\_q$.peak();
10:    if mem_hoder is invalid then
11:      $mems \leftarrow fpga\_channel$.drain_out();
12:      foreach $mem \in mems$ do
13:        $full\_batch\_q$.push_back($mem$);
14:      $mem\_hoder \leftarrow free\_batch\_q$.pop();
15:      $cmd \leftarrow$ cmd_generator($batch\_size$);
16:      foreach $index \in batch$ do
17:        $file \leftarrow data\_collector$.getNextFile();
18:        $cmd$.encode($file$.metainfo, $index$, $mem\_hoder$.phy
           _addr);
19:      $mems \leftarrow fpga\_channel$.submit($cmd$);
20:      foreach $mem\_item \in mems$ do
21:        $full\_batch\_q$.push_back($mem\_item$);
22:  $mem\_pool$.recycle();
23:  $fpga\_channel$.recycle();
24:  return
```

Algorithm 2. Memory Managing and Dispatching

```
1: initMemoryPool ($size$, $counts$)
2:   $baseAddr \leftarrow$ getHugePage ($counts * size$)
3:   foreach $index \in [0, counts]$ do
4:     $item.size \leftarrow size$
5:     $item.phyAddr \leftarrow baseAddr + index * size$
6:     $item.virtAddr \leftarrow$ phy_to_virt ($item.phyAddr$)
7:     $mem\_pool.free\_batch\_queue$.push ($item$)
8:   return $mem\_pool$
9: Dispatching ($solvers$)
10:  foreach $solver$ in$solvers$ do
11:    $cpu\_item \leftarrow full\_batch\_queue$.blocking_pop()
12:    $free\_tq \leftarrow solver.Trans\_Queues[FREE]$
13:    $gpu\_item \leftarrow free\_tq$.blocking_pop()
14:    GPUMemcpyAsync($gpu\_item.deciveAddr$, $cpu\_item.virt$
       $Addr$, $solver.copyStream$)
15:    $working\_queue[CPU]$.push_back($cpu\_item$)
16:    $working\_queue[GPU]$.push_back($gpu\_item$)
17:  //recycle mem. buffers by synchronizing streams
18:  foreach $gpuSolver$ in$Solvers$ do
19:    GPUStreamSync($solver.copyStream$)
20:    $cpu\_item \leftarrow working\_queue[CPU]$.pop()
21:    $gpu\_item \leftarrow working\_queue[GPU]$.pop()
22:    $solver.Trans\_Queues[FULL]$.push($gpu\_item$)
23:    $free\_batch\_queue$.push($cpu\_item$)
24:  return
```

4.2 Host Bridger Design

To efficiently manage the decoder in FPGAs and the computing engines in GPUs, the host bridger provides two abstractions (i.e., decoding abstraction and memory management) in DLBooster. We present them in detail as follows.

4.2.1 Decoding Abstraction

To provide flexible programmability and mask underlying operations on hardware to common users, DLBooster implements an FPGA decoder abstraction (i.e., *FPGAReader*) in the host bridger. Inside *FPGAReader*, an *FPGAChannel*, which is composed of a *task queue* and a *completion queue*, is built for the communication between the decoder in FPGAs and *FPGAReader*. Besides, a *DataCollector* is built to maintain the metainfo of raw data. To efficiently manage multiple FPGA devices, each *FPGAReader* is bound to an FPGA decoder with a unique *FPGAChannel* and isolated from the others. The *DataCollector* is shared globally by all *FPGAReaders*.

4.2.2 Memory Management

Note that FPGAs cannot directly operate on the virtual address of host memory. Therefore, a memory mapping mechanism (e.g., *mmap*) is required to translate memory addresses (virtual → physical) for the FPGA decoder. To achieve high decoding performance, the FPGA decoder processes data in batches. However, *mmap* cannot allocate such a contiguous memory block with the large size required by the FPGA decoder. Therefore, DLBooster redesigns a memory mapping mechanism based on Linux *HugePage* [48] to manage a large block of contiguous memory.

As described in Algorithm 2 (lines 1–8), DLBooster implements a memory pool to manage memory buffers at runtime and offers an abstraction for memory access through two queues (i.e., *Free_Batch_Queue* and *Full_Batch_Queue*). Initially, a huge block (1 GB by default) of memory with consecutive addresses is allocated to DLBooster. Then, the large memory block is sliced into small blocks, which are put into the *Free_Batch_Queue* for next time use. There are three entries

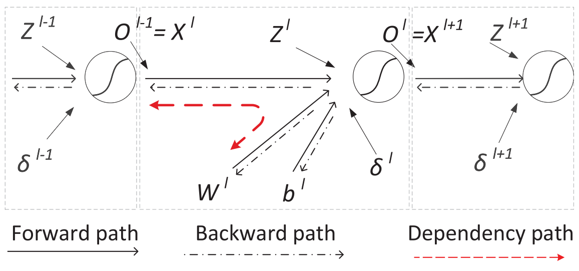


Fig. 4. DNN structure and dependency chain.

(i.e., virtual address, physical address and block size) in each block to record its identification.

At runtime, memory blocks flow between the two queues and act as high-speed carriers that move the data processed by FPGA decoders from host memory to GPU computing engines. Each memory block is first sunk into the FPGA decoder via a decoding *cmd*. Loading a batch of processed data, it is then inserted into *Full_Batch_Queue*. On its other side, the *Dispatcher* simultaneously moves the processed data on each memory block to a GPU device, ending up with inserting the block into *Free_Batch_Queue* for next time use. The *Dispatcher* asynchronously moves data from memory pool to multiple GPU devices (shown as the `Dispatching` in line 9 of Algorithm 2).

5 Optimizing BP Computation Workflow

Nowadays, BP is the most widely used algorithm to train DNNs, which follows the chain rule (shown as Fig. 4) with strict dependencies. Training a DNN with BP can be performed in two steps: namely, a *forward* pass, in which the training data are forwarded through the NN layer by layer to compute a loss ($\mathcal{L}$), and a *backward* pass, where the loss is propagated in a reversed order to compute gradients (∇W, ∇b) of parameters (W, b) in each layer.

5.1 Partial Dependency in BP Workflow

As shown in Fig. 4, there is a partial dependency between the computational workloads of training NNs with BP. We take the example of how gradients are computed on a layer to show the details. In the *backward* stage (refer to the *Backward Path* in Fig. 4), there are two main computing events to be launched in each layer, namely, propagating the loss (δ) to its previous layer (Eq. (1)) and computing gradients for the weighted kernel (Eq. (2)) and bias (Eq. (3)) on the current layer. From Equations (1), (2), and (3) and Fig. 4, we can find that the computing events on each layer require the loss (δ) from its neighboring layer (upper) and the data (summarized in Table 2) on the current layer, which means that the loss from upper layers has already been propagated to the current layer before those computing events are launched.

When training NNs on GPUs, the computing engines in DL frameworks (e.g., Caffe, MxNet, TensorFlow, etc.) usually organize computing tasks by groups[2] (or layers) and asynchronously submit them to GPUs via CUDA streams for parallel execution [5], [21], [34], [45]. Computing engines would synchronize those events on CUDA streams before submitting new ones for the next layer to handle the dependency problem. Considering the lightweight workloads on small layers (or training with a small batch size), such coarse-grained submission and synchronization strategies usually fail to fully utilize the computing resources of GPUs in training, leaving the potential optimization as follows.

TABLE 2
Notions of the *Forward/Backward* Pass When Running BP on DNNs

symbol	description
b^l	Bias in layer l.
W^l	Weighted kernel in the l^{th} layer.
X^l	$X^l = O^{l-1}$, is the input feature map in the l^{th} layer.
Z^l	$Z^l = X^l \odot W^l + b^l$ is the temporary output feature map of layer l, where $\odot$ is a operator (*conv.* for Conv. layer and *mul* for FC layer).
O^l	$O^l = \sigma\left(Z^l\right)$ is the output feature map of layer l.
$\sigma(\cdot)$	Activation function. It gives nonlinear transformation for the temporary output Z^l to the real output O^l at layer l, i.e. $O^l = \sigma\left(Z^l\right)$
$\mathcal{L}$	$\mathcal{L} = \frac{1}{2}\|\|\hat{y} - y\|\|_2^2$ is the loss of NN, where y is the ground truth and $\hat{y}$ is the predicted value of the NN.
δ^l	Loss on the l^{th} layer. $\delta^l = \frac{\partial \mathcal{L}}{\partial Z^l}$

$$\begin{aligned}\delta^{l-1} &= \frac{\partial \mathcal{L}}{\partial Z^{l-1}} = \frac{\partial \mathcal{L}}{\partial Z^l}\cdot\frac{\partial Z^l}{\partial X^l}\cdot\frac{\partial X^l}{\partial Z^{l-1}} \\ &= \begin{cases}\left(\left(W^l\right)^T\cdot\delta^l\right)\odot\sigma'\left(Z^{l-1}\right) & FC\ layer \\ \delta^l\cdot rot180\left(W^l\right)\odot\sigma'\left(Z^{l-1}\right) & Cov.\ layer\end{cases}\end{aligned} \tag{1}$$

$$\nabla W^l = \frac{\partial \mathcal{L}}{\partial W^l} = \frac{\partial \mathcal{L}}{\partial Z^l}\cdot\frac{\partial \mathcal{L}}{\partial W^l} = X^l\cdot\delta^l \tag{2}$$

$$\nabla b^l = \frac{\partial \mathcal{L}}{\partial b^l} = \frac{\partial \mathcal{L}}{\partial Z^l}\cdot\frac{\partial Z^l}{\partial b^l} = \delta^l\cdot I. \tag{3}$$

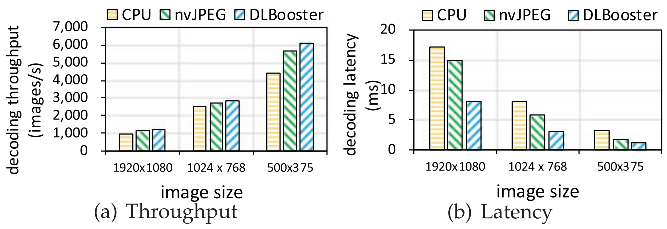
5.2 Overlapping Computation on Different Layers

Revisiting Eqs. (1), (2), and (3), we find that the computation for each layer only relies on the loss (δ^l) propagated from the upper layer and the variables (W^l, X^l) on the current layer. Since parameters on each layer are already updated in the *forward* pass, and there is no dependency between the parameter's gradients (∇W^l, ∇b^l) on the current layer and the loss (δ^{l-1}) propagated to the previous layer, we can launch the gradient computing events for the current layer once the loss from the upper layer is ready. Such a mechanism leaves us the chance to improve GPU utilization by overlapping[3] the workloads of computing the loss to the previous layer and computing gradients of the parameters on the current layer.

As described in Algorithm 3, we reorganize computing events of loss propagating and gradients computing on the GPU. Specifically, we implement a *main BP thread*, which

2. Although many DL frameworks provide low-level DL operators to describe the fine-grained computing tasks, modern DNNs are usually built using high-level DL operators provided by cuDNN and cuBLAS to achieve high performance on GPUs.

3. From the training view, the computation of BP for each layer is still bound by the dependency chain. Therefore, such overlapping designs do no harm to the convergence of training



(a) Throughput (b) Latency

Fig. 5. Experiment results of data preprocessing on different backends.

runs on a high-priority CUDA stream [37] (lines 2 – 10), to handle the loss propagating job. Once the loss has been propagated to the lth layer, the *main BP thread* computes the loss for the $l-1\text{th}$ layer, ending up with notifying the *BP assist threads* to compute the gradients of parameters on the lth layer. Running with high priority, the NN's loss is rapidly propagated from the output layer to the first layer and will not be affected by other workloads on the GPU. From the views of *BP assist threads*, when a new gradient computing task is submitted by the *main BP thread*, meaning that the computational requirements of this task are met, it will be dispatched to a normal CUDA stream to be executed according to the workloads on the GPU (lines 12 – 17).

Algorithm 3. Scheduling the Computational Workloads in the GPU When Running BP on DNNs

```
1:  For main BP thread:
2:  foreach iter inIterations do
3:    $net$.sync_and_wait_gradient ( )
4:    $net$.Forward($first\_layer$, $last\_layer$)
5:    for ($l = last\_layer$; $l > first\_layer$; $l$- -) do
6:      $stream \leftarrow HIGH\_PRIO\_CUDA\_STREAM$
7:      $layer[l]$.BP_over_X($stream$)
8:      if layer[l] requires gradient then
9:        $assist\_thread\_channel$.push($l$)
10:   $assist\_thread\_channel$.push($DONE$)
11: For assist BP threads:
12: while not exit do
13:   foreach $l$ in$assist\_thread\_channel$ do
14:     if $l$ is not$DONE$ then
15:       $layer[l]$.BP_over_W($CUDA\_STREAM$)
16:       $layer[l]$.BP_over_b($CUDA\_STREAM$)
17:     $reduce\_q$.push($l$)
```

Such an overlapping design allows GPUs to achieve higher utilization, mitigating the impacts of imbalance of workloads on different layers and the dependencies between them. The benefit of such overlaps becomes even more distinct when we use smaller batch sizes, because the workload of small-batch training is more unlikely to saturate GPUs and leaves more room for the optimization.

6 Implementation and Evaluation

We implement a DLBooster prototype based on the designs in previous sections, which are summarized as follows:

(1) Using OpenCL [47], we build a prototype of the FPGA decoder for image processing tasks, where we scale the Huffman decoding and resizing units to 4 ways and 2 ways respectively in an Intel Arria 10 GX [11] FPGA device according to their workloads and hardware constraints.

(2) We put all the decoder logic in a mirror (FPGA IP core) which can be freely downloaded to FPGA devices.

(3) We optimize each component in DLBooster and expose simple APIs, with which DLBooster is easily integrated into different DL frameworks with trivial effort ($\sim$220 lines of code for both TensorRT [32] and NVCaffe [33]) to provide high-performance online data preprocessing services.

(4) We enable the fine-grained scheduling for the BP workflow in NVCaffe by re-implementing corresponding APIs (i.e., `Net::Backward` and `Layer::Backward`) for the training, following the design in Section 5.

Putting all together, we evaluate DLBooster with micro benchmarks and two typical real-world DL tasks to demonstrate its potential to speed up DL applications.

6.1 Micro Benchmark

In this section, we evaluate how each design (i.e., data preprocessing offloading and BP workflow scheduling) of DLBooster performs through micro benchmarks as follows.

6.1.1 *Data Preprocessing*

To evaluate the performance (throughput and processing latency) of DLBooster on image preprocessing,[4] we conduct the experiment with the following settings: (1) In the CPU-based backend, we implement the decoding logic based on OpenCV [27] and enable 16 Intel Xeon E5 2630-v3 (16 hyper threads in all) cores to decode at runtime; (2) In nvJPEG, we dispatch decoding instances onto multiplex CUDA streams on a NVIDIA Tesla P100 GPU. (3) In DLBooster, we scale the Huffman decoding and resizing units to 4 ways and 2 ways respectively in an Intel Arria 10 GX [11] FPGA device.

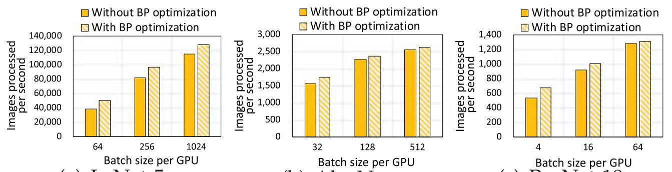
Figs. 5a and 5b present the throughput and latency results of image decoding respectively. We conclude that: (1) Compared with the other two baselines, DLBooster achieves competitive decoding throughput, showing the potential of offloading decoding workloads to FPGAs. (2) DLBooster achieves the lowest decoding latency because of directly fetching data from devices without CPU involved, while nvJPEG involves twice data moving (disk $\rightarrow$ host memory $\rightarrow$ GPU memory) before decoding, which introduces the nontrivial latency. This experiment shows the efficiency of DLBooster on image preprocessing with FPGAs.

6.1.2 *BP Computational Workflow Scheduling*

To evaluate the efficiency of scheduling computational workloads when training NNs with BP, we conduct this experiment of training typical NNs (i.e., LeNet-5, ResNet-18 and AlexNet) on 1 P100 GPU. We use NVCaffe as the computing engine and train NNs with synthetic data to avoid the impacts of data preprocessing. We compare the image processing throughput when training NNs with different batch size settings (for different GPU utilization).

As shown in Fig. 6, (1) there are additional 5–15 percent training performance gains when enabling the scheduling, showing its effectiveness. (2) The gained training

4. We take data loading (from an NVMe disk), image decoding, and resizing (to the shape: $224\times224\times3$) into account, as they are all essential parts of data preprocessing.



(a) LeNet-5 (b) AlexNet (c) ResNet-18

Fig. 6. Throughput Comparison when training different NNs on NVCaffe with enabling scheduling optimization and the baseline.

throughput is significant when the batch size is small, where the GPU is not fully utilized. When the batch size grows larger, the performance gaps decrease. It's because the increasing computational workloads of CUDA kernels can utilize more GPU cores, and the training bottleneck has been migrated to the GPU hardware. However, our optimization can still achieve a few performance gains in this case.

6.2 Evaluation on Real-World DL Applications

We use two kinds of end-to-end DL workflows (i.e., local training (Section 6.2.2) and online inference (Section 6.2.3)) to show the efficiency of DLBooster in accelerating DL applications.

6.2.1 Experimental Setup

Testbed Configuration. We run the experiments on a GPU server with: two Intel Xeon E5-2630-v3 CPUs (32 hyper threads in all), two NVIDIA Tesla P100 GPUs, an FPGA device (Arria 10 GX [11]) and an NVMe disk (Optane 900p [49]) both from Intel, a 40 Gbps NIC and 64 GB DDR3 DRAM. As for the software, the programs are built on CentOS-7 with the latest third-party dependencies, such as NVIDIA CUDA-v9.0, NCCL-v2.4, cuDNN-v7.13, etc.

Computing Engines & Data Preprocessing Backends. In training experiments, we integrate the CPU-based backend, LMDB and DLBooster with NVCaffe-v0.17.3 [33]. As for the inference, we use the CPU-based backend, nvJPEG and DLBooster to offer data preprocessing services in TensorRT-v4.0 [32]. Each backend is configured with well-tuned settings.

Models and Datasets. In the experiments, we evaluate DLBooster with real-world DL applications and datasets, i.e., LeNet-5 [50] (with MNIST [51]), ResNet-18 [19] and AlexNet [2] (with ILSVRC2012 [39]) for training, and VGG-16 [18], ResNet-50 [19] and GoogLeNet [52] for inference, respectively. MNIST [51] includes 60,000 grayscale images, and ILSVRC2012 [39] consists of 1.28 million color images.

6.2.2 Local Training Experiment

To demonstrate how DLBooster benefits training DNNs on GPU clusters, we conduct the experiments of training LeNet-5, ResNet-18 (with FP16) and AlexNet with different data preprocessing backends. To better show the contribution of DLBooster on the end-to-end DL workflows, we only enable the BP optimization for the NVCaffe with DLBooster.

Throughput. Fig. 7 presents the training throughput comparison, in which we find that: (1) DLBooster enables NVCaffe to approach the training performance roof of GPU when training different NNs; (2) LMDB allows NVCaffe to achieve high throughput when training NNs with one GPU. However, it degrades the training throughput by 30 percent when training AlexNet with 2 P100 GPUs (Fig. 7c), limited by its poor data caching design. (3) With the optimization of BP workflow, DLBooster boosts the additional training performance of NVCaffe by up to 14 percent in some cases.

CPU Overhead. Fig. 8 presents the CPU costs of the training experiment. We conclude that: (1) There are around 1.5 cores consumed by DLBooster when training all three DNNs, whereas LMDB consumes around 2.5 cores in the training. (2) The CPU-based backend consumes much more CPU cores: each GPU requires about 12 cores and 7 cores in the training of AlexNet and ResNet-18 respectively. This experiment has shown the potential of DLBooster to save CPU/GPU cores when training NNs in GPU clusters.

6.2.3 Online Inference Experiment

To demonstrate the potential of DLBooster to boost inference tasks,[5] we infer VGG-16, ResNet-50 and GoogLeNet by TensorRT with different online data preprocessing backends (i.e., CPU-based backend, nvJPEG [24] and DLBooster) and observe the latency, throughput and CPU costs respectively. We launch 5 TCP clients to send images between two servers connected by 40Gbps networks to reproduce the online inference workflow. The images ($500 \times 375 \times 3$ on average) in the JPEG format are randomly selected from ILSVRC2012.

Latency. For simplicity and fairness, we only focus on the time costs introduced by data preprocessing and NNs inferring. Fig. 10 presents the results of latency comparison.

We find that: (1) TensorRT with DLBooster has the lowest latency (1.2 *ms*), compared with the CPU backend (3.4 *ms*) and nvJPEG (1.8 *ms*). (2) As the batch size increases, the time costs of inference with all three backends significantly vary, due to the increasing computational workloads in TensorRT. (3) The latency in nvJPEG-enabled TensorRT is low when using a small batch size and becomes much higher when using a large batch size. This is because both nvJPEG and TensorRT require a lot of computation resources, resulting in the competition for GPU cores between each other.

Throughput and CPU Overhead. Figs. 9 and 11 present the results of throughput and CPU costs respectively. Similar to the training experiment, (1) DLBooster achieves the highest inference throughout and the lowest processing latency, and Both nvJPEG and the CPU-based backend suffer from great performance degradation, consuming a lot of CPU/GPU cores. (2) Compared with DLBooster, nvJPEG allows TensorRT to achieve only 1/3 image inferring throughput due to the competition problem mentioned before. (3) By contrast, the CPU-based backend consumes more than 20 CPU cores to process data at runtime, which is more than that in the training experiment. This is because: (i) Inferring NNs is much faster than the training. (ii) The decoding efficiency decreases due to runtime overheads when so many CPU cores run simultaneously, such as context switching, lock problem, etc. DLBooster not only achieves the highest image inferring throughput but also runs in an efficient

5. Note that inference only contains the computation in the forward pass, the BP optimization of DLBooster do not work here

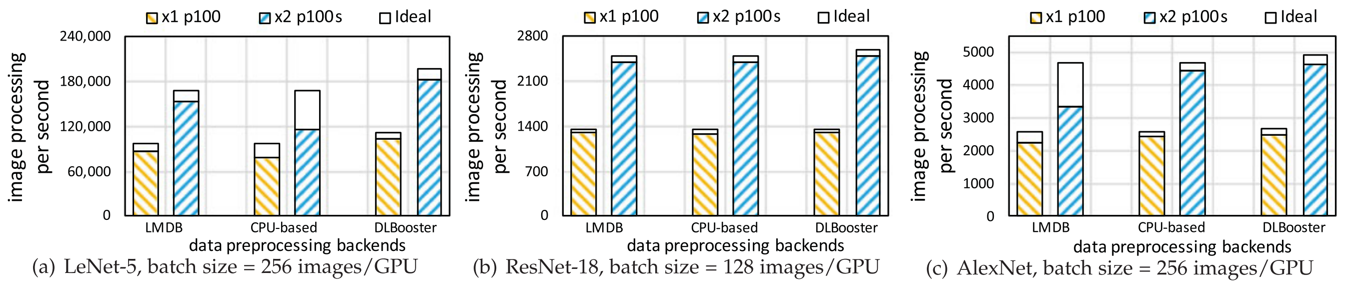


Fig. 7. Throughput of training NNs by NVCaffe with different backends. Each experiment is configured with optimal settings. In DLBooster, the optimization for BP is enabled for NVCaffe (resulting in higher training performance bound). ResNet18 is trained with FP16.

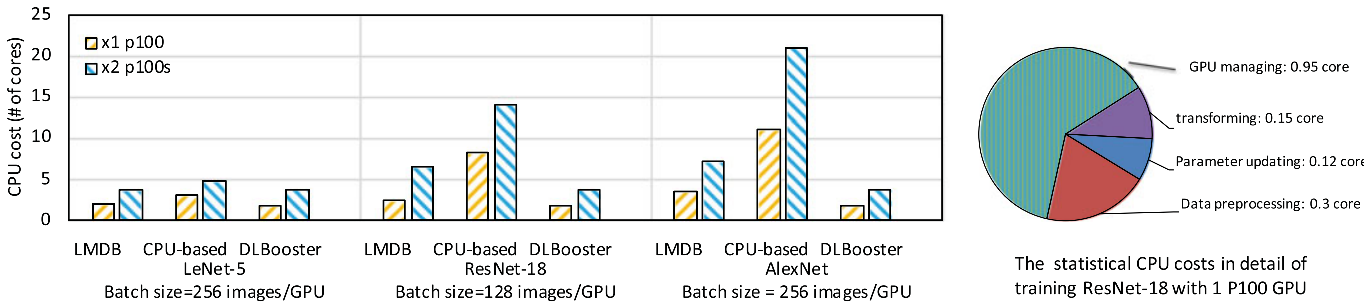


Fig. 8. CPU costs in the training experiment: The left shows the CPU costs when training LeNet-5, ResNet-18 and AlexNet with three typical data preprocessing backends (LMDB, the CPU-based backend, and DLBooster). The right shows the CPU costs in detail of training ResNet18 with the DLBooster backend, where around 1.5 CPU cores are consumed in all and 0.3 core is used to process the training data.

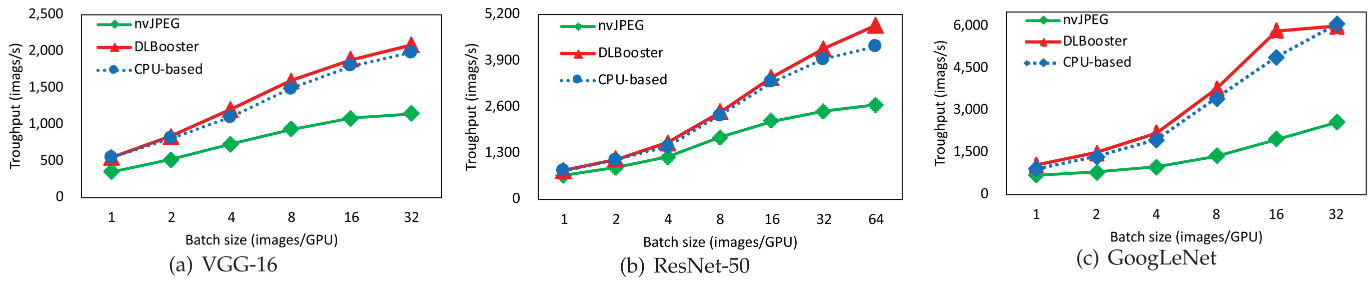


Fig. 9. Throughput comparison of inferring VGG-16, ResNet-50, and GoogLeNet on TensorRT with different data preprocessing backends (i.e., the CPU-based backend, nvJPEG, and DLBooster). FP16 is enabled to speed up the inference of ResNet-50.

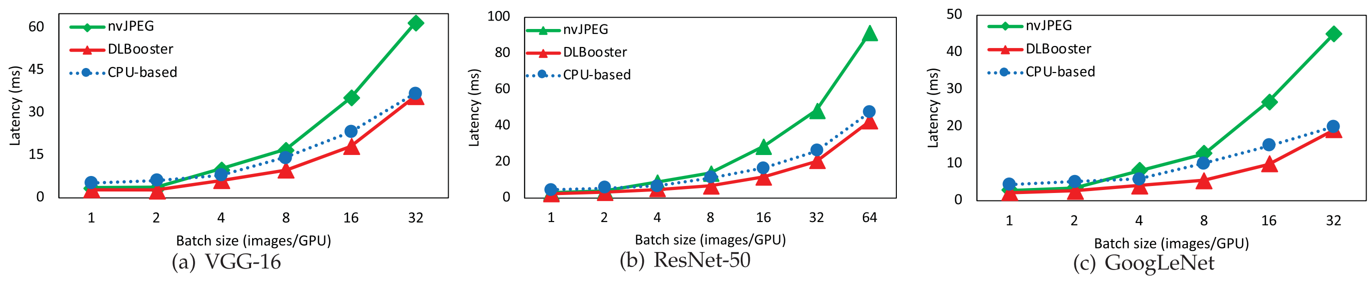


Fig. 10. Latency comparison of inferring VGG-16, ResNet-50, and GoogLeNet on TensorRT with different data preprocessing backends (i.e., the CPU-based backend, nvJPEG, and DLBooster). FP16 is enabled to speed up the inference of ResNet-50.

way: it consumes less than 1 CPU core to manage the FPGA decoder at runtime.

Summary. In this section, we have evaluated DLBooster on both micro benchmarks and real-world image processing tasks and datasets. Compared with existing data preprocessing backends, DLBooster can not only improve the image processing throughput by 1.4× – 2.5× in both training and inference workflows, but also reduce 1/3 processing latency in online image inferring tasks. Moreover, it can save as many as 90 percent CPU cores in some cases. DLBooster shows the potential to speed up DL applications in the cloud.

7 Discussion and Experience

We have demonstrated the potential of DLBooster to accelerate DL applications in previous sections. In this section, we analyze the benefits and concerns brought by DLBooster.

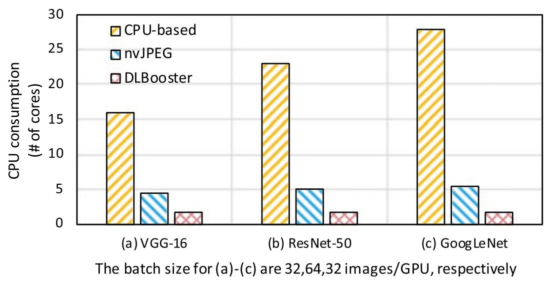


Fig. 11. CPU cost in the inference experiments.

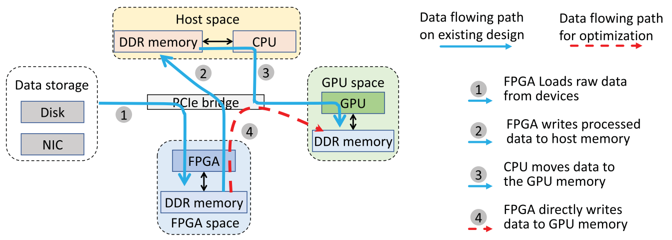

Host space
DDR memory
CPU
Data storage
Disk
NIC
PCIe bridge
GPU space
GPU
DDR memory
FPGA
DDR memory
FPGA space
Data flowing path on existing design
Data flowing path for optimization
1 FPGA Loads raw data from devices
2 FPGA writes processed data to host memory
3 CPU moves data to the GPU memory
4 FPGA directly writes data to GPU memory

Fig. 12. Further optimizations of data flowing path for DLBooster: FPGA bypasses CPU and directly writes the processed data to GPU memory.

(1) *Programming Flexibility*

DLBooster codesigns hardware with software, and the biggest concern about DLBooster is the programming flexibility with FPGAs. However, from our experience, programming with FPGAs has become easier with the prosperity of FPGA ecosystems. Particularly, we build the FPGA decoder with OpenCL [47] in our experiment, which is as flexible as the C language in terms of programming. Furthermore, (i) we design DLBooster with open interfaces to simplify the integration of DLBooster into popular DL frameworks. (ii) The decoding kernel in the FPGA decoder is pluggable and can be easily replaced with other decoding kernels. (iii) We are going to extend DLBooster with more decoding kernels for different DL applications, such as NLP [3] and speech recognition [42]. We will keep optimizing DLBooster to simplify its deployment in production.

(2) *Economic Benefits*

DLBooster optimizes data preprocessing with FPGAs and benefits both cloud users and cloud service providers when more DL applications are deployed in the cloud [4].

To Cloud Users. DLBooster leverages one FPGA device to provide decoding services with the same performance as 20 CPU cores. According to our studies, FPGAs are widely deployed and sold at a low price in the cloud [53], while CPUs and GPUs are much expensive. For example, in Azure [26], a physical CPU core (2 hyper threads) sells for \$ 0.11 per hour [26] or brings as much as \$ 900 revenue per year.

To Cloud Providers. Today, CPUs, GPUs and FPGAs are deployed at scale in the cloud, such as Azure. The daily costs of maintaining such large-scale devices are expensive. On the one hand, the power consumption of FPGAs (~25W [11]) is much lower than GPUs (~250W [9]) and CPUs (~130W [26]). Therefore, migrating some heavy workloads from CPUs/GPUs to FPGAs can reduce the power consumption. On the other hand, deploying DLBooster at scale can save a mass of CPU cores, which brings significant revenue to the cloud service providers.

(3) *Further Data Path Optimization*

Currently, DLBooster involves the HugePage memory in user space to hold the processed data from the FPGA decoder, which are then moved to GPUs for use (shown as Fig. 12). However, such design will also introduce extra overheads of CPU consumption and processing delay. To deal with this challenge, our recent work, DUA [53], has shown the exploration of making data center resources such as the GPU available to FPGAs. In DLBooster, we can further extend the FPGA decoder to directly write the processed data to GPU devices, in which we can further reduce the overhead of CPU costs and time delay. We will keep optimizing DLBooster in terms of both FPGA decoder performance and decoding extensions to simplify its deployment in the cloud.

8 Related Work

Data Preprocessing Designs for DL. Many DL frameworks have provided their data preprocessing backends to facilitate DL applications. In general, they can be categorized into two types. The first is *offline primitive* that processes datasets first and reloads them at need, such as *LMDB* [20], *RecordIO* [23], *TFRecord* [22], etc. The second is *online primitive* which burns volumes of CPU/GPU cores to decode data at runtime. DALI [25] is a hybrid high-performance data preprocessing backend, which exploits both GPU-based nvJPEG [24] and CPU cores to decode images at runtime for DL applications. Other efficient data preprocessing solutions have also been explored. For example, some researchers [15] split the dataset into multiple high-speed NVMe disks and load them in parallel at runtime. Different to their designs, DLBooster exploits FPGAs to offer satisfying data preprocessing services in an efficient way.

Computation Workflow Optimization for DL. Many research works [14], [15], [16], [36], [46] have discussed how to boost DL applications by fully utilizing hardware resources. TVM [35] and TC [54] are the end-to-end DL compilers to generate fused kernels by exploring optimal scheduling strategies on different hardware backends. In general, they mainly focus on the inference, while DLBooster optimizes the BP workflow of training NNs on GPUs. ByteScheduler [55] and TicTac [56] are two representative communication scheduling schemes for the DML training. In general, they rely on the scheduling of computation graphs in DML frameworks, which do not well distinguish the error propagating computation and gradients computing workloads during the training. DLBooster implements the fine-grained scheduling to better overlap the computation while promising the computation on the critical path first.

FPGA-Based Acceleration for DL. With the advantages of low power consumption and ultralow processing latency, FPGAs have been used to speed up emerging workloads such as video processing [57], deep learning [28], [58], etc. In general, many existing FPGA-accelerated DL studies are about the explorations of FPGAs on accelerating the specific computational workloads for fast inference of NNs, such as CNN [59], [60], LSTM [61], [62], etc. Differently, DLbooster improves the end-to-end DL workflows by offloading some key data preprocessing workloads to FPGAs.

9 Conclusion

In this work, we first dive into the end-to-end DL workflow and show the distinct bottlenecks of data preprocessing. Furthermore, we propose DLBooster, which offloads some heavy decoding workloads to FPGAs, to improve the data preprocessing performance in emerging DL applications. Additionally, we optimize the BP workflow by rescheduling the computational workloads with the partial dependency to improve the hardware utilization when training NNs on GPUs. Our experiments on typical real-world image processing tasks and datasets demonstrate that compared with baselines, DLBooster can improve the image processing throughput by up to $1.4\times$ – $2.5\times$ and shorten the inference latency by 1/3. In the future, we will further optimize DLBooster to speed up more DL applications in the cloud.

Acknowledgments

The authors would like to appreciate Jiaxin Lin (Beihang University & Microsoft Research), Xi Fan, Xinyi Yu (Shanghai Jiao Tong University & Microsoft Research), Lei Qu, Ran Shu, and Peng Cheng (Microsoft Research) for their valuable contributions to this work. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800500, in part by the Research and Development Program in Key Areas of Guangdong Province under Grant 2018B010113001, and in part by the National Natural Science Foundation of China under Grant 61772305. An earlier version [1] of this work has been presented as a conference article at the 48th International Conference on Parallel Processing (ICPP 2019).

References

[1] Y. Cheng *et al.*, "DLBooster: Boosting end-to-end deep learning workflows with offloading data preprocessing pipelines," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–11.
[2] A. Krizhevsky *et al.*, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[3] J. Devlin, M.-W. Chang, Kenton Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv: 1810.04805*.
[4] I. Nucleus Research, "TensorFlow on AWS," 2018. [Online]. Available: https://d1.awsstatic.com/whitepapers/nucleus-tensorflow-2018.pdf
[5] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[6] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[7] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, vol. 16, pp. 265–283.
[8] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
[9] Introduction of NVIDIA DGX-2 GPU cluster, Accessed: 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-2/
[10] G. Cloud, "AI and machine learning platform in Google cloud," 2019. [Online]. Available: https://cloud.google.com/ai-platform/
[11] Intel, "Intel arria-10 FPGAs," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html
[12] Mellanox, "An overview ethernet cards in mellanox," 2019. [Online]. Available: http://www.mellanox.com/page/ethernet_cards_overview
[13] B. Yi *et al.*, "Towards zero copy dataflows using RDMA," in *Proc. SIGCOMM Posters Demos*, 2017, pp. 28–30.
[14] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes," 2017, *arXiv: 1711.04325*.
[15] P. Goyal *et al.*, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," 2017, *arXiv: 1706.02677*.
[16] Y. You *et al.*, "ImageNet training in minutes," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–10.
[17] M. Cho, U. Finkler, S. Kumar, D. Kung, V. Saxena, and D. Sreedhar, "PowerAI DDL," 2017, *arXiv: 1708.02188*.
[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[19] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1026–1034.
[20] F. Zou, "Creating ImageNet LMDB in Caffe," Accessed: 2020. [Online]. Available: https://github.com/intel/caffe/wiki/How-to-create-Imagenet-LMDB
[21] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[22] M. Daoust *et al.*, "Introduction to TFRecords," Accessed: 2020. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tf-records
[23] A. Acharya *et al.*, "Image transforms and recordio file creation of MXNet," 2018. [Online]. Available: https://cwiki.apache.org/confluence/display/MXNET/Image+Transforms+and+RecordIO+file+Creation, 2018.
[24] nvJPEG: GPU-accelerated JPEG decoder, Accessed: 2020. [Online]. Available: https://developer.nvidia.com/nvjpeg
[25] NVIDIA DALI and NVIDIA nvJPEG, Accessed: 2020. [Online]. Available: https://news.developer.nvidia.com/announcing-nvidia-dali-and-nvidia-nvjpeg
[26] D. Firestone *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 51–64.
[27] Introduction to OpenCV, Accessed: 2020. [Online]. Available: https://opencv.org/
[28] E. Chung *et al.*, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar./Apr. 2018.
[29] M. Feldman, "Microsoft goes all in for FPGAs to build out ai cloud," TOP500 supercomputer sites. 2017. [Online]. Available: https://www. top500. org/
[30] D. Pellerin, "FPGA accelerated computing using AWS F1 instances," *AWS Public Sector Summit*, 2017.
[31] G. Leopold, "Intel, Facebook accelerate datacenters with FPGAs," Accessed: 2020. [Online]. Available: https://www.enterpriseai.news/2016/03/23/intel-facebook-accelerate-datacenters-fpgas/
[32] NVIDIA, "NVIDIA tensorrt programmable inference accelerator," 2018. [Online]. Available: https://developer.nvidia.com/tensorrt
[33] NVCaffe, Accessed: 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/caffe/
[34] Introduction to CUDA toolkit, Accessed: 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[35] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 578–594.
[36] A. O. Ashari, "On optimizing machine learning workloads via kernel fusion," *ACM SIGPLAN*, vol. 50, no. 8, pp. 173–182, 2015.
[37] Introduction to CUDA stream, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html
[38] Y. LeCun *et al.*, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[39] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.
[40] J. F. Gemmeke *et al.*, "Audio set: An ontology and human-labeled dataset for audio events," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2017, pp. 776–780.

[41] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman, "Reading text in the wild with convolutional neural networks," *Int. J. Comput. Vis.*, vol. 116, no. 1, pp. 1–20, 2016.

[42] L. Deng *et al.*, "Recent advances in deep learning for speech research at Microsoft," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, vol. 26, pp. 8604–8608.

[43] L. Bottou *et al.*, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*, 2010, pp. 177–186.

[44] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.

[45] Introduction to cuDNN, Accessed: 2020. [Online]. Available: https://developer.nvidia.com/cudnn

[46] S. Rajbhandari *et al.*, "Optimizing CNNs on multicores for scalability, performance and goodput," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 267–280, 2017.

[47] J. E. Stone *et al.*, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, 2010.

[48] Huge page support in linux kernel, Accessed: 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[49] Intel optane SSD 900p series, Accessed: 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series.html

[50] Y. LeCun *et al.*, "Comparison of learning algorithms for handwritten digit recognition," in *Proc. Int. Conf. Artif. Neural Netw.*, 1995, vol. 60, pp. 53–60.

[51] Y. LeCun *et al.*, "The MNIST database of handwritten digits," Accessed: 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[52] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[53] R. Shu *et al.*, "Direct universal access: Making data center resources available to FPGA," in *Proc. 16th USENIX Conf. Netw. Syst. Des. Implementation*, 2019, pp. 127–140.

[54] N. Vasilache *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv: 1802.04730*.

[55] Y. Peng *et al.*, "A generic communication scheduler for distributed DNN training acceleration," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.

[56] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," 2018, *arXiv: 1803.03288*.

[57] S. Wang *et al.*, "Live video analytics with FPGA-based smart cameras," in *Proc. Workshop Hot Topics Video Analytics Intell. Edges*, 2019, pp. 9–14.

[58] K. Ovtcharov *et al.*, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Res. WhitePaper, vol. 2, no. 11, pp. 1–4, 2015.

[59] C. Zhang *et al.*, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proc. Int. Symp. Low Power Electron. Des.*, 2016, pp. 326–331.

[60] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[61] S. Cao *et al.*, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 63–72.

[62] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 75–84.