ARTICLES

Journal of Semiconductors
(2020) 41, 022403
doi: 10.1088/1674-4926/41/2/022403

Towards efficient deep neural network training by FPGA-based batch-level parallelism

Cheng Luo[1, †], Man-Kit Sit[2], Hongxiang Fan[2], Shuanglong Liu[2], Wayne Luk[2], and Ce Guo[2]

[1]State Key Laboratory of ASIC and System, Fudan University, Shanghai 200050, China

[2]Department of Computing, Imperial College London, London, United Kingdom

Abstract: Training deep neural networks (DNNs) requires a significant amount of time and resources to obtain acceptable results, which severely limits its deployment in resource-limited platforms. This paper proposes DarkFPGA, a novel customizable framework to efficiently accelerate the entire DNN training on a single FPGA platform. First, we explore batch-level parallelism to enable efficient FPGA-based DNN training. Second, we devise a novel hardware architecture optimised by a batch-oriented data pattern and tiling techniques to effectively exploit parallelism. Moreover, an analytical model is developed to determine the optimal design parameters for the DarkFPGA accelerator with respect to a specific network specification and FPGA resource constraints. Our results show that the accelerator is able to perform about 10 times faster than CPU training and about a third of the energy consumption than GPU training using 8-bit integers for training VGG-like networks on the CIFAR dataset for the Maxeler MAX5 platform.

Key words: deep neural network; training; FPGA; batch-level parallelism

Citation: C Luo, M K Sit, H X Fan, S L Liu, W Luk, and C Guo, Towards efficient deep neural network training by FPGA-based batch-level parallelism[J]. *J. Semicond.*, 2020, 41(2), 022403. http://doi.org/10.1088/1674-4926/41/2/022403

1. Introduction

Deep neural networks (DNNs) have achieved remarkable achievements on various demanding applications including image classification[1, 2], object detection[3, 4] and semantic segmentation[5, 6]. In resource-limited settings, the development of real-time and low-power hardware accelerators is especially critical, and hence various hardware devices including FPGAs and ASICs have been utilized for implementing embedded DNN applications. In particular, FPGAs are gaining popularity because of their capability to provide superior energy efficiency and low-latency processing while supporting high reconfigurability, making them suitable for accelerating rapidly evolving deep neural networks[7–9].

However, most of the existing FPGA accelerators are designed for inference with low-precision DNN models, which are trained on high-precision models (e.g. 32/64-bit floating point models) separately on GPU or CPU. Since DNNs employ different precision formats for training and inference, they often need further fine-tuning to achieve acceptable accuracy. The separate training/inference processes make existing FPGA accelerators difficult to support, for example, systems requiring continual learning[10]. Various low-precision training techniques including mixed precision[11, 12], fixed-point[13, 14] and ternary[15, 16] parameters, have been proposed to reduce the fine-tuning overhead by low-precision models.

In this paper, we explore the benefits and drawbacks of employing CPU, GPU and FPGA platforms for low-precision training. An novel FPGA framework is developed to support DNN training on a single FPGA with a low-precision format of 8-bit integer (int8). Our objective is to determine if the fine-grained customizability and flexibility offered by FPGAs can be exploited to outperform cutting-edge GPUs in low precision training in terms of speed and power consumption.

To meet our objective, the following challenges should be addressed.

(1) The training process, compared to inference process, brings additional computations and different operations performed in backward propagation[17]. This leads to differences in requirements for hardware architecture and computational resources.

(2) Existing FPGA accelerators for inference usually exploit image-level and layer-level parallelism for efficient computing. On contrast, FPGA accelerators for training need to proceed with batches of training examples in parallel. Therefore, effective exploitation of the batch-level parallelism should contribute significant acceleration.

(3) Throughput is the primary performance metric of concern for training, while inference is latency sensitive. This cause batch-level parallelism to be neglected at inference accelerators.

To solve these problems, this paper proposes a novel FPGA architecture for DNN training by introducing a batch-oriented data pattern which we refer to as channel-height-width-batch (CHWB) pattern. The CHWB pattern allocates training samples of different batches at adjacent memory addresses, which enables parallel data transfer and processing to be achieved within one cycle. Our architecture can support the entire training process inside a single FPGA and accelerate it with batch-level parallelism. A thorough exploration of the design space with different levels of parallelism and their corresponding architectures with respect to resource consumption and performance is also presented in this paper.

Moreover, we propose DarkFPGA, an FPGA-based deep

Correspondence to: C Luo, 16110720014@fudan.edu.cn
Received 7 NOVEMBER 2019; Revised 19 DECEMBER 2019.

learning framework with a dataflow architecture. Our approach is built on Darknet framework[18], a neural network framework written in C and CUDA, with FPGA implementation written in MaxJ[19]. The proposed paper is an extended version work from an earlier conference paper[20]. Contributions of the previous version were as follows:

(1) A novel accelerator for a complete DNN training process. A dataflow architecture that explores batch-level parallelism for efficient FPGA acceleration of DNN training is developed, providing a power-efficient and high-performance solution for efficient training.

(2) A deep learning framework for low-precision training and inference on FPGAs called DarkFPGA. We perform extensive performance evaluations for our framework on the MAX5 platform for the training of several well-known networks.

(3) An automatic optimization tool for the framework to explore the design space to determine the optimal parameters for a given network specification.

Additionally, this paper contributes as follows:

(1) Toward the timing problems caused by batch-level parallelism, the pipelining registers are inserted to reduce fan-out, while the super-logic region allocation is proposed to avoid long-wires interconnection.

(2) Training with INT8 weights, instead of ternary weights, is proposed to maintain stable training performance for low-precision model.

The organization of this paper is organized as follows. Section 2 reviews the training and inference processes and some existing FPGA-based accelerators. Section 3 introduces the deep learning algorithm training using low-bits number system. Section 5 proposes the dataflow accelerators designed for GEMM operations. Section 6 discusses the design space exploration for optimizing accelerator design. Section 7 presents our framework of DarkFPGA. Section 8 shows the experimental results, and we conclude the whole paper on Section 9.

2. Background

This section provides a background information of DNN training, emphasizing its difference from inference. Meanwhile, the cutting-edge FPGA accelerators for deep neural network are also introduced here.

2.1. Training versus Inference

The training consists of forward propagation to compute the loss of the cost functions, and backward propagation to compute the gradients of the cost function, subsequently using gradients to update the model weights for learning desirable behavior. Unlike inference with only forward propagation, training with backward propagation is more computationally expensive and introduce additional operations for backward propagation.

Fig. 1 illustrates the overview of the inference and training of a convolutional layer. For a specific layer l, the inference process with forward propagation simply convolves the input activations (A_l) with the weights (W_l) to generate the output activations for the next layer (A_{l+1}). On contrast, the training process separately performs forward propagation to compute the errors using the loss function, and backward propagation to convolve the errors (E_{l+1}) from the last layer with the current weights (W_l) to calculate the errors to be propagated to the previous layer (E_l). The backward propagation of training also compute the gradients (G_l) with respect to the loss function using multiply–accumulate operation (MAC). These gradients update the current weights according to the chosen optimization algorithm like Adam[21].

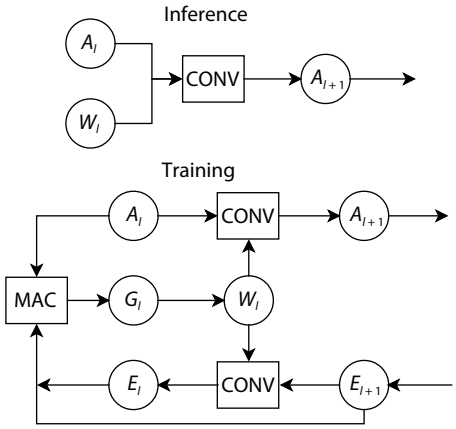


Fig. 1. A overview of inference and training processes on the convolutional layer.

Algorithm 1: Pseudocode for training convolutional layers

```
1 Forward propagation:
2 for b = 1 to B do
3   for c = 1 to C × K do
4     for f = 1 to F do
5       for im = 1 to H*W do
6         A_{l+1}[b][f][im] += W_l[f][c] *A_l[b][c][im]
7 Backward propagation:
8 for b = 1 to B do
9   for c = 1 to C × K do
10    for f = 1 to F do
11      for im = 1 to H*W do
12        E_l[b][c][im] += W_l[f][c] *E_{l+1}[b][f][im]
13 Gradient Generation:
14 for b = 1 to B do
15   for c = 1 to C × K do
16     for f = 1 to F do
17       for im = 1 to H*W do
18         G_l[b][f][c] += A_l[b][c][im] *E_{l+1}[b][f][im]
```

For better understanding, the pseudocode for training a convolutional layer is presented on Algorithm 1, which provides a precise description for the training process. The meaning of the notations can be found in Table 1, where the same set of notation is also followed in the rest of this paper.

2.2. Related works

Most FPGA accelerators mainly focus on the DNN inference acceleration[22–27]. They[24, 28] usually exploits image-level and layer-level parallelism extensively for efficient inference speedup. For training accelerators[29–35], Geng *et al.*[31] explore layer-level parallelism for training a model on multiple FPGAs in a pipelined manner. Li *et al.*[32] study different reconfigurable communication patterns on a multi-FPGA cluster. Dicecco *et al.*[33] study low-precision training with a reduced precision floating-point library. However, those accelerators usually deploy the inference architecture naively for training without considering the batch-level parallelism and additional backward operations, which may lead to undesirable performance.

Table 1. Parameters for FPGA training.

Parameter	Description
B	the batch size of training examples
C	the size of channel
F	the size of filter
K	the kernel size of weights
H	the height of frames
W	the width of frames

Recently, some researchers[29, 36] attempt to tackle the problem by distributing the DNN computations across a heterogeneous FPGA-CPU system. Moss *et al.*[36] propose to perform the core GEMM operations on FPGAs and leave CPU for the remaining jobs. This solution works well on FPGA-CPU heterogeneous system but requires effective load balancing support for heterogeneous devices, since unpredictable communication cost between CPUs and FPGAs can make the FPGA-CPU cross communication a new bottleneck of the design. Based on our profiling in Section 8, the operations executed on CPU may require more computational time than FPGA acceleration of matrix multiplication.

With the objective of speeding up training, this paper studies the acceleration of entire training on a single FPGA, explores the parallelism in training batches, and provides an architecture suitable for bidirectional propagation. We propose a low-precision DNN training framework accelerated on a single FPGA platform. Compared to other frameworks, our proposed customizable FPGA design achieves about 10 times speedup over a CPU-based implementation and is about 2.5 times more energy efficient than a GPU-based implementation.

3. Low-precision DNN training algorithm

Our low-precision training algorithm is developed based on WAGE[15], which is modified version for better FPGA implementation using shift-based linear mapping and hardware-friendly quantization. Our optimizations are illustrated here.

The basic idea of WAGE[15] is to constrain four operands to low-bitwidth integers: weight W and activation A in forward propagation, error E and gradient G in backward propagation, using corresponding quantization operations QW, QA, QE, QG in computation flow to reduce precision. Experiments show stable accuracy can be obtained on multiple datasets. However, for hardware implementation, complex mathematical functions including logarithm, exponential operation which can be easily realized on CPU/GPU are hardly mapped on FPGA-implementation. Therefore, hardware-friendly customized operations are necessary for the accurate and efficient FPGA-based deep neural network training.

3.1. Shift-based linear mapping

In order to quantize floating-point numbers to fixed-point number, k-bit linear mapping is adopted on WAGE[15], where continuous and unbounded values are discretized with uniform distance $\sigma(k)$:

$$\sigma(k) = 2^{(1-k)}, \quad k \in N_+,$$
$$Q(x,k) = \mathrm{Clip}\{\sigma(k) \times \mathrm{round}\left[\frac{x}{\sigma(k)}\right], -1+\sigma(k), 1-\sigma(k)\}.$$

Here round function maps quantized floating-point number to nearest fixed-point number. Clip is the saturation function to clip unbounded values to $[-1+\sigma(k), 1-\sigma(k)]$.

Considering large hardware implementation overhead for floating-point operations, mathematical equivalent integer operations are introduced in our implementation, where the linear mapping is transformed into shifting from large data format (32-bit integers) to small integers (k-bit integers) which can be expressed as:

$$\sigma(k) = 2^{(k-1)}, \quad k \in N_+,$$
$$Q(x,k,\mathrm{shift}) = \mathrm{Clip}\{(x + \mathrm{round\_value}) \gg \mathrm{shift}, -1+\sigma(k), 1-\sigma(k)\},$$
$$\mathrm{round\_value} = 1 \ll (\mathrm{shift} - 1).$$

Here we replace division operations used in float-point equations d with shift operations with an additional monolithic scaling factor shift for shifting values distribution to an appropriate order of magnitude. The scaling factor shift is obtained in WAGE[15] by following equation.

$$\mathrm{shift}(x) = \mathrm{round}(\log_2 x).$$

With complex logarithm and exponential operation, the shift(x) requires extensive resources to be implemented on FPGA. To handle this problem, we fine-tune this formula, which is used to obtain the nearest power-of-two value from input x, to obtain ceiling power-of-two value as follow:

$$\mathrm{shift}(x) = \mathrm{ceil}(\log_2 x).$$

After fine-tuning, the shift factor is obtained from smallest power-of-two value greater than x, and can be re-expressed by bit-wise operations as follow:

$$\mathrm{shift}(x) = (\mathrm{leading1}(x) + 1).$$

Here leading1 function detects the position of the most significant "important" bit and return the index of the most significant "important" bit only. After detailed experiments, the fine-tuning has no effect on the convergence of network training but more hardware-friendly for FPGA implementation.

3.2. Quantization details

The quantization operations consist of four operations QW, QA, QE, QG. Theses operations is responsible to quantize four training operands including weight W, activation A, error E and gradient G to low-bitwidth format.

3.2.1. Weight QW

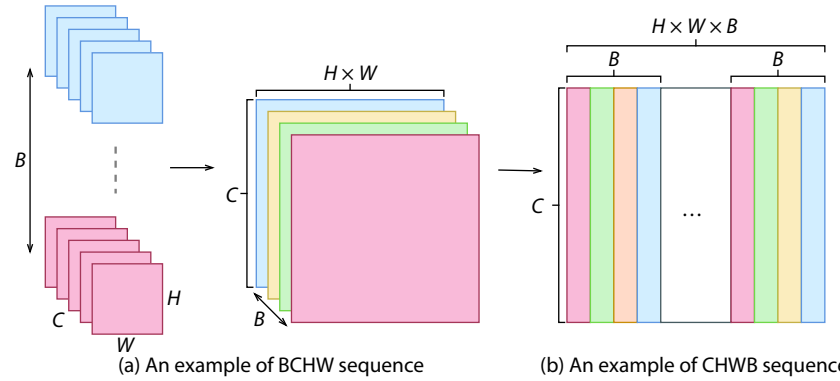
Weights are initialized on software platform based on the initialization method of He *et al.*[37], which can be formulated as:

$$W\ U(-L,+L),\ L = \max\{\sqrt{6/n_{\mathrm{in}}}, L_{\mathrm{min}}\},\ L_{\mathrm{min}} = 1,$$

where n_{in} is the layer fan-in number, and the original limit $\sqrt{6/n_{\mathrm{in}}}$ is calculated to keep same variance between inputs and outputs of the same layer theoretically. The additional limit L_{min} is a minimum value that the uniform distribution should reach.

3.2.2. Activation QA

For activation, the bitwith of activation would increase after computation. A filter-wise scaling factor a_{shift} is intro-



(a) An example of BCHW sequence (b) An example of CHWB sequence

Date	0	1	2	3	0	1	2	3	...
DRAM Addr	× 0	× 1	× 2	× 3	× 4	× 4	× 6	× 7	...

(c) BCHW sequence in DRAM space

Date	0	0	0	0	1	1	1	1	...
DRAM Addr	× 0	× 1	× 2	× 4	× 5	× 6	× 7	× 8	...

(d) CHWB sequence in DRAM space

Fig. 2. (Color online) Comparison of BCHW and CHWB patterns.

duced for shifting values distribution to an appropriate order of magnitude, which can be obtained by following function:

$$a_{\text{shift}} = \log_2(\max\{\text{shift}(L_{\min}/L), 0\}).$$

This factor is pre-defined constant for each layer determined by the network structure. Using this factor we can obtain the quantized activation using the following equation:

$$a_{\text{q}} = Q(a, k_{\text{A}}, a_{\text{shift}}).$$

3.2.3. Error QE

Experiments from WAGE[15] demonstrate the orientation of errors plays an important role on the converge performance during training. Therefore, orientation-based quantization scales errors into [−1,1] by dividing a shift factor as:

$$e_{\text{q}} = Q(e, k_{\text{E}}, \text{shift}(\max|e|)),$$

where $\max|e|$ extracts the layer-wise maximum absolute value among errors. Since the shift value extracts the smallest power-of-two value for $\max|e|$, an obvious optimization method is using "or" operations instead of max to improve the hardware performance.

$$e_{\text{q}} = Q(e, k_{\text{E}}, \text{shift}(\text{or}|e|)),$$

where $\text{or}|e|$ executes the bit-wise or operation on the layer-wise maximum absolute value among error.

3.2.4. Gradient QG

Since we only preserve the relative value of the error after shifting, the gradients are shifted consequently. Here we first rescale the gradient g with another scaling factor:

$$g_{\text{q}} = \text{Bernoulli}\{(\eta \times g) \gg g_{\text{shift}}\},$$
$$g_{\text{shift}} = \text{shift}(\text{or}|g|)),$$

where η is learning rate which is constrained as power-of-two. So η can also be represented by the corresponding shift value. Here gradients are quantized as errors by the bit-wise operation $\text{or}|g|$ instead of the maximum function $\max|g|$.

Bernoulli[38] function was originally design in floating-point number system to stochastically sample fractional parts to either 0 or 1. The nature of the Bernoulli distribution is the larger number has higher probability to 1 and the smaller number has higher probability to 0. On contrast in integer system, Bernoulli function is stochastically rounding the shifting parts of quantized value, which is realized by a random number generator MersenneTwister, a widely-used general-purpose pseudorandom number generator[39] to generate Limited range of random numbers according to shift data with uniform distribution. The MersenneTwister adds Bernoulli property by addition as following equation:

$$g_{\text{q}} = \text{Clip}\{(\eta \times g + \text{round\_value}) \gg g_{\text{shift}}, -1 + \sigma(k), 1 - \sigma(k)\},$$
$$g_{\text{shift}} = \text{shift}(\text{or}|g|)),$$
$$\text{round\_value} = \text{random\_int} \bmod (1 \ll g_{\text{shift}}),$$

where random_int is random numbers of 32-bit integer format.

4. Date pattern and tilling technique

4.1. CHWB Pattern

For DNN training, the weights, activations, errors and gradients are too large to be stored completely in the on-chip memory, where only a portion of data can be cached on-chip while the remaining is kept off-chip. As the bandwidth between the on-chip and off-chip memory is limited, exploring an optimal data access pattern to for efficient bandwidth utilization is necessary for training.

Currently, the most widely-used data pattern for training on GPUs is referred as batch-channel-height-width (BCHW), which depicts the order of data dimensions in the memory space[40], where the elements along the lowest dimension W are stored consecutively. An example of data represented in the BCHW pattern is shown on Fig. 2(a), whose corresponding data layout is illustrates in the DRAM in Fig. 2(c). But this pattern is difficult to fetch the elements from different batches in burst mode, because they are usually not stored consecutively in memory. Therefore, BCHW data pattern may under-utilize the bandwidth when exploring batch-level parallelism.

To handle the problem, we develop the channel-height-width-batch (CHWB) pattern to explore batch-level parallelism without compromising bandwidth utilization on FPGAs. As shown in Fig. 2(b), the elements from adjacent batches are

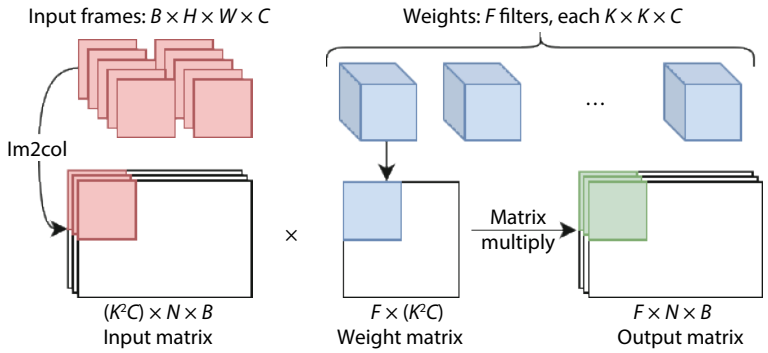


Fig. 3. (Color online) The tiling flow for convolution.

allocated consecutively, which allows the memory interface to simultaneously read multiple training examples. In this manner, CHWB data pattern enables our accelerator to acquire all necessary input data with a single DRAM burst access, and greatly improve bandwidth utilization for FPGA accelerator.

4.2. Tiling

Tiling is a common optimization technique to improve bandwidth utilization for DNN acceleration on resource-limited FPGA devices[41]. The strategy partitions large input frames into smaller tiles of data, where each tile can be fitted into the on-chip memory of an FPGA. For training with some resource-intensive tasks such as matrix transpose, tiling strategy is necessary for their FPGA implementation.

For the CHWB pattern, we consider tiling along four data dimensions: batch tile T_B, channel tile T_C, filter tile T_F and image tile T_I, which correspond to the size of a tile along the dimension. Consider the input matrix transpose between the image dimension and channel dimension, as well as the weight matrix transpose between the channel dimension and filter dimension during training[42], the image tile T_I, the channel tile T_C and the filter tile T_F are all set to same value. In this case, two levels of parallelism are explored in our design: the batch-level parallelism P_B and the image-level parallelism P_I, which are controlled by the tiling parameters T_B and T_I respectively.

Taking convolution to explain how tiling technique works. In Fig. 3, the input matrix is stored using CHWB pattern 3-dimensions $T_B \times T_I \times T_I$ tiled blocks, while the weight matrix is stored with two-dimensional $T_I \times T_I$ tiled blocks. In the forward propagation, the tiled input matrix and the tiled weight matrix are sequentially multiplied to obtain one output submatrix.

The tiling parameters T_B and T_I must be chosen carefully, since a larger tile size can improve performance by reducing the iterations of data transfers, but more on-chip memory resources are required for storing larger tiles. In Section 6. we will explore the optimization of the tile parameters on resource-limited FPGA.

5. FPGA accelerators for DNN training

In this section, we follow the idea of CHWB pattern and tiling technique to develop the architecture of our training accelerator.

5.1. System overview

A system overview of our FPGA-based training accelerator is presented on Fig. 4, which consists of a computation kernel, a global controller and a DDR controller for off-chip memory transfer. The computational kernel consists of a batch splitter, a set of processing elements (PEs) and a batch merger. When a stream of training batches arrives at the kernel, the splitter divides the stream into multiple parallel streams via shift registers to facilitate batch-level parallelism. The streams are then processed by the PEs in parallel. Each PE involves a general matrix multiplication kernel (GEMM kernel) or an auxiliary kernel to perform training operations. After processing, the streams are merged into a single output stream, then sent to the DDR controller. The global controller is responsible for controlling the behaviour of each computation kernel, including assigning memory addresses for loading/writing data through the DDR controller, enabling special operations required by particular layers, and controlling the direction of the data flow. The CPU sets the network configuration in the global controller before starting training.

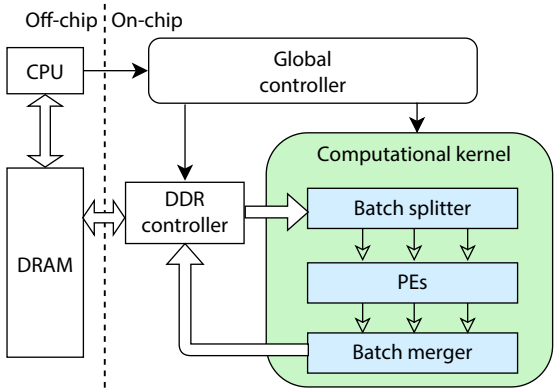


Fig. 4. (Color online) System overview.

5.2. Unified GEMM kernel

Fig. 5 presents the architecture of the GEMM kernel, which provides a unified datapath to support the convolutional and fully-connected computations of the forward and backward propagation, as well as the gradient generation. This unified approach employ matrix multiplication to implement for these computations, where only the input/output matrix to/from the kernel needs to be changed. Therefore, we can avoid time-consuming dynamic reconfiguration[29] or using separate kernels for different operations[31].

Before any computation, the input data streams are stored in the input buffers, which are organized as a double buffer in order to overlap the data transfer and matrix transposition with the computation. As shown in Fig. 6, when the

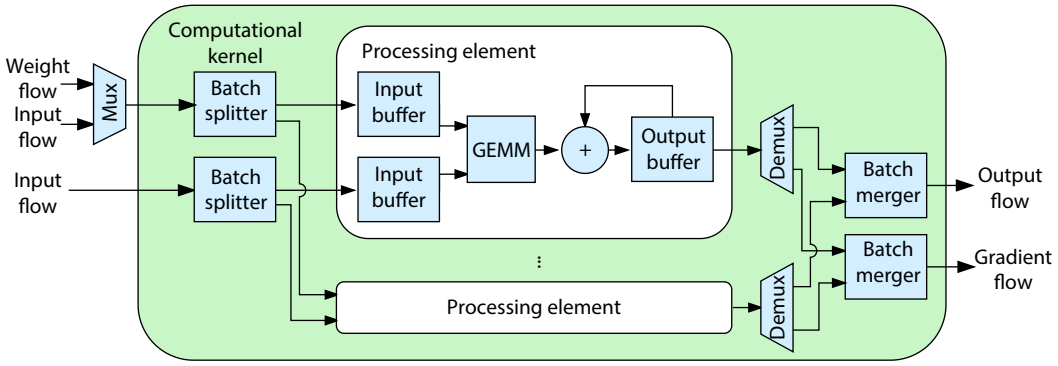


Fig. 5. (Color online) Hardware architecture of GEMM kernel.

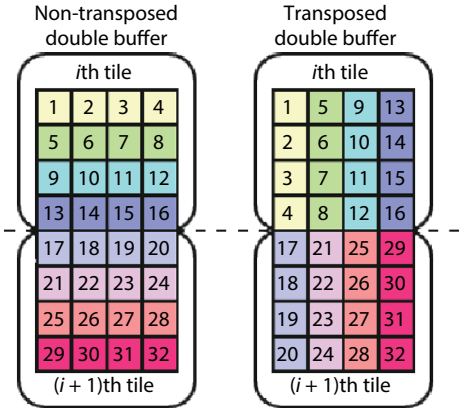


Fig. 6. (Color online) Input double buffer supporting matrix transposition.

$(i+1)$th tile flows from the batch splitter to the input buffer, the ith tile is sent to the GEMM kernel for the computation. Note that the weight stream bypasses the batch splitter and enters the input buffers directly, as the weight data are shared by different tiles across the same batch.

The GEMM kernel fetches data from the input buffers to perform tiled matrix multiplication. The intermediate values during each iteration are stored in the output buffers for the next iteration. The final results are post-processed by the batch merger then transferred back to the DRAM. The details of the tiled matrix multiplication are shown in Algorithm 2. Noted that the shift factor of forward propagation is predefined while the the shift factor of backward propagation is obtained from output results. Therefore, the quantization function can be attached after forward propagation to reduce output bandwidth. On contrast, the output results maintain Int32/Int16 format during backward propagation as well as gradients generations, which require quantization with the help of auxiliary kernels.

In order to support different modes of operations for the forward propagation, the backward propagation and the gradient generation, the global controller dynamically re-configures the buffers and data flow on the datapath. Under the control of global controller, the input buffer can be configured to perform on-the-fly matrix transposition for the computation of backward propagation. Furthermore, the multiplexer can be switched to feed the different input streams to their corresponding processing elements, and the demultiplexer can be switched to direct the output stream to the appropriate post-processing unit.

Algorithm 2: Pseudocode of tiled matrix multiplication

1 Consider that the weight matrix and gradient matrix are transferred into $T_I \times T_I$ tiled blocks, where input frames, output frames and error frames are transferred as 3-dimensions $T_B \times T_I \times T_I$ tiled blocks. In particular, the input frames and output frames of fully-connected layer are transferred as 2-dimensions $T_B \times T_I$ tiled blocks
2 Convolutional forward propagation:
3 for $f = 1$ *to* F/T_I **do**
4 **for** $im = 1$ *to* $(H * W)/T_I$ **do**
5 **for** $b = 1$ *to* B/T_B **do**
6 **for** $c = 1$ *to* $C \times K/T_I$ **do**
7 $A_{l+1}(b)(f, im)_c = W_l(f, c) \times A_l(b)(c, im)$
8 $A_{l+1}(b)(f, im) \mathrel{+}= A_{l+1}(b)(f, im)_c$
9 Quantize $(A_{l+1}(b)(f, im))$
10 Output $A_{l+1}(b)(f, im)$
11 Convolutional backward propagation:
12 for $c = 1$ *to* $C \times K/T_I$ **do**
13 **for** $im = 1$ *to* $(H * W)/T_I$ **do**
14 **for** $b = 1$ *to* B/T_B **do**
15 **for** $f = 1$ *to* F/T_I **do**
16 $E_{l-1}(b)(c, im)_f = W_l(f, c) \times E_l(b)(f, im)$
17 $E_{l-1}(b)(c, im) \mathrel{+}= E_{l-1}(b)(c, im)_f$
18 Output $E_{l-1}(b)(c, im)$
19 Convolutional gradients generations:
20 for $f = 1$ *to* F/T_I **do**
21 **for** $c = 1$ *to* $C \times K/T_I$ **do**
22 **for** $b = 1$ *to* B/T_B **do**
23 **for** $im = 1$ *to* $(H * W)/T_I$ **do**
24 $G_l(b)(f, c)_{im} = E_{l+1}(b)(f, im) \times A_l(b)(c, im)^T$
25 $G_l(b)(f, c) \mathrel{+}= G_l(b)(f, c)_{im}$
26 $G_l(f, c) \mathrel{+}= \sum_{b=1}^{T_B} G_l(b)(f, c)_{im}$
27 Output $G_l(f, c)$
28 Fully-connected forward propagation:
29 for $f = 1$ *to* F/T_I **do**
30 **for** $b = 1$ *to* B/T_B **do**
31 **for** $c = 1$ *to* C/T_I **do**
32 $A_{l+1}(b)(f)_c = A_l(b)(c) \times W_l(f, c)^T$
33 $A_{l+1}(b)(f) \mathrel{+}= A_{l+1}(b)(f)_c$
34 Output $A_{l+1}(b)(f)$

5.3. Auxiliary kernels

The auxiliary kernels accelerate supplementary operations with batch-level parallelism including im2col, col2im, max-pooling, reshape, summation, nonlinear functions and quantization as well as their backward counterparts (if neces-

sary). These supplementary operations processed independently since they have no learnable weights and occupy only a small amount of total computation.

For various supplementary operations, various types of separate processing units is implemented to support them. In particular, the maxpooling units are responsible for computing the maximum value and corresponding index over a number of neighbor pixels, whereas the backward maxpooling units propagate errors the to chosen index of subgraphs. The im2col expands the input feature map into column vectors, and col2im accumulate column the vectors back to input feature map. The quantization units are designed for casting intermediate variables of errors and gradients generated during backward propagation and gradients generation to low bitwidth quantized data (8-bit integer in our design). The summation units simply add two flows together and the reshape units change the location of data in buffers.

6. Design space exploration

This section presents the design space exploration for optimizing the proposed DNN training accelerator. The performance of FPGA implementations is affected by factors including batch tiling size T_B, image tiling size T_I and bitwidth L for training. The bitwidth is pre-defined while the two tiling sizes are decided by our optimization model. To maximize performance, we develop bandwidth modelling, resource modelling and performance modelling to enable design space exploration.

6.1. Resource modeling

There are three kinds of hardware resources in FPGAs: LUT, Block RAM and DSP, which form the resource constraints of our design space. We present equations to estimate the utilization for each of them.

First, the resource consumption of the global controller and the DRAM controller is independent of the design parameters, while they are defined as $\mathrm{LUT}_{\mathrm{fix}}$, $\mathrm{DSP}_{\mathrm{fix}}$, $\mathrm{BRAM}_{\mathrm{fix}}$.

Second, the resource consumption of the computational kernels is affected significantly by different design parameters. For example, BRAMs are utilized in the input buffers of GEMM kernels and their usage is given by:

$$\mathrm{BRAM} = \frac{4 \times T_B \times T_I^2 \times (2 \times L_I) + 4 \times T_I^2 \times L_W}{\mathrm{BRAM}_{\mathrm{SIZE}}},$$

where the constants 4 are contributed by the double buffers for both the normal matrix and the transposed matrix.

The multiply-and-add units utilize the DSPs as

$$\mathrm{DSP} = T_B \times T_I \times D_{\mathrm{mul}} + T_B \times A_I \times D_{\mathrm{add}} + T_B \times D_{\mathrm{add}},$$

where D_{mul} and D_{add} are the DSP usage of the multiplier and adder. A_I is the level of tree adder in the computational kernel, which equals to $\log_2(T_I)$.

Finally, an approximate regression model is proposed to estimate the resource consumption of LUT as it is difficult to predict statically:

$$\mathrm{LUT} = T_B \times T_I \times \beta + T_B \times \delta,$$

where β, δ are linear function parameters pre-trained based on a specific platform.

6.2. Bandwidth modeling

There are three streams flowing from the DRAM to the GEMM kernels. In each cycle of convolution, one weight is read from the weight stream while T_B input values are read from input frame stream. The results are accumulated in processing elements before N iterations of the convolution are completed. When it comes to the fully-connected layer, additional T_I weights are needed which increase the bandwidth requirement. Therefore, the theoretical maximum bandwidth requirements for computing the convolutional and fully-connected layers with frequency f are:

$$\mathrm{BW}_{\mathrm{CONV}} = (T_B \times L_I + \frac{T_B \times T_I}{N} \times L_O + L_W) \times f,$$

$$\mathrm{BW}_{\mathrm{FC}} = (T_B \times L_I + \frac{T_B \times T_I}{N} \times L_O + T_I \times L_W) \times f,$$

where L_I, L_O, L_W are the bitwidth of the input frame, the output frame and the weight stream.

For the auxiliary kernel, the bandwidth requirements are relatively large compared to the small amount of computations performed. In general, it may take one or two input values to generate one or two output values, which handles up to 4 values in each cycle. As these operations benefit from batch-level parallelism, the bandwidth requirements have also multiplied T_B times:

$$\mathrm{BW}_{\mathrm{auxiliary}} = (2 \times T_B \times L_I + 2 \times T_B \times L_O) \times f.$$

6.3. Performance modeling

In each clock cycle, a GEMM kernel can accomplish T_I matrix multiplication operations. Therefore for our batch-parallel architect with T_B kernels, the total computational time under frequency f is:

$$T_{\mathrm{CONV}} = \frac{B \times C \times K \times F \times H \times W}{T_I \times T_B \times f},$$

$$T_{\mathrm{FC}} = \frac{B \times C \times F}{T_I \times T_B \times f}.$$

However, the above formulae are only valid for the sequential case. In fact, in order to support parallel computing, tiled matrices are filled with zero values which may affect the actual computational time. Therefore, the computational time is estimated as:

$$T_{\mathrm{CONV}} = \frac{\lceil B \rceil^{T_B} \times \lceil C \times K \rceil^{T_I} \times \lceil F \rceil^{T_I} \times \lceil H \times W \rceil^{T_I}}{T_B \times T_I \times f},$$

$$T_{\mathrm{FC}} = \frac{\lceil B \rceil^{T_B} \times \lceil C \rceil^{T_I} \times \lceil F \rceil^{T_I}}{T_B \times T_I \times f},$$

$$\lceil X \rceil^T = \mathrm{ceil}(X/T) \times T,$$

where function $\lceil X \rceil^T$ means mapping X to the least multiple of T greater than or equal to X, which indicates that these formulae are identical only when tiling sizes are set as factors of the corresponding parameters.

On the other hand, in each cycle of the auxiliary kernel, frames batches can be handled simultaneously, where the computational time of auxiliary kernels is:

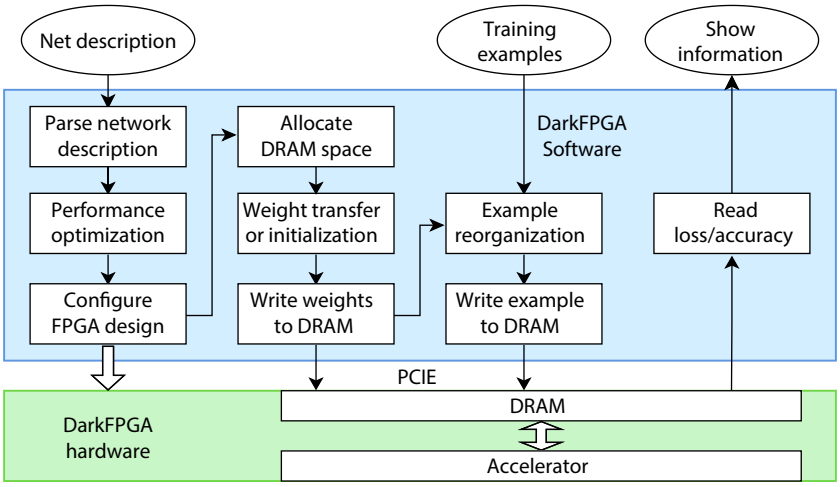


Fig. 7. (Color online) The DarkFPGA framework.

$$T_{\text{auxiliary}} = \frac{\lceil B \rceil^{T_B} \times C \times H \times W}{T_B \times f},$$

Benefited from our dataflow architecture, the transmission time of the computational kernels can be overlapped by the communication time.

By evaluating the performance of every combination based on the above models, a single-objective optimization tool can be built for minimal execution time as:

$$\text{Minimize Time} = T'_{\text{CONV}} + T'_{\text{FC}} + T'_{\text{auxiliary}},$$

$$\text{where} \begin{cases} \text{LUT} + \text{LUT}_{\text{fix}} \leqslant \text{LUT}_{\text{limit}}, \\ \text{BRAM} + \text{BRAM}_{fix} \leqslant \text{BRAM}_{\text{limit}}, \\ \text{DSP} + \text{DSP}_{\text{fix}} \leqslant \text{DSP}_{\text{limit}}, \\ \text{BW} \leqslant \text{BW}_{\text{limit}}, \end{cases}$$

where $\text{LUT}_{\text{limit}}$, $\text{BRAM}_{\text{limit}}$, $\text{DSP}_{\text{limit}}$, BW_{limit} are limited on-chip resources, and $T'_{\text{CONV}} + T'_{\text{FC}} + T'_{\text{auxiliary}}$ are the expected computation time for a specific network description.

7. The DarkFPGA framework

For our proposed dataflow architecture, we present DarkFPGA, a hardware/software co-designed FPGA framework for effective training. DarkFPGA framework is built with scalable accelerator architecture which is software-definable to support various DNN networks and different parallel levels through deploying different FPGA bitstreams, where a multi-level parallelism scalable FPGA design is developed. Moreover, a optimizing tool is included to produce optimised design for optimised performance based on user constraints.

We automate the process of exploring design parameters for the DarkFPGA framework, which accelerates the entire training process with a unified module on FPGA. Our tool can receive a network description and a training dataset to produce the most suitable parameters for the accelerator. The overview of our DarkFPGA framework and optimizing tool are illustrated in Fig. 7 with six stages:

(1) Parse network description. The tool predicts optimized parameter values and selects a suitable FPGA bitstream to configure hardware.

(2) Allocate device DRAM space for the activations, weights, errors and gradients.

(3) Initialize weights and transfer them to DRAM.

(4) Fetch and transfer the training samples to DRAM. Data reorganization is used to convert training samples into the CHWB sequence.

(5) Launch FPGA acceleration.

(6) Train neural network iteratively. Transfer loss and accuracy information back to the host for each complete training batch.

8. Experimental result

We evaluate our framework on the Maxeler MAX5 platform, which consists of a Xilinx ultrascale+ VU9P FPGA. Three 16 GB DDR4 DIMMs are installed on the platform with a maximum bandwidth of 63.9 GB/s. Our hardware accelerator works at 200 MHz. Maxcompiler 2019.1 and Vivado 2018.3 are used for synthesis and implementation. The VGG-like network[43] trained on the Cifar10[1] dataset is evaluated in the following training experiments. Differ form the conference version of this paper[20], we use INT8 weights instead of ternary weight during training. Although the use of INT8 weights cause inevitable degradation in training performance, it is able to obtain more stable accuracy performance for training.

Noted while our implementation are able to achieve the massive parallelization with dataflow architecture, it may have difficulty in making the timing closure for a high clock frequency, or even passing the place and route. This is partly because these direct interconnects become long wires when the DSP blocks are distributed among the whole FPGA chip, and large-scale data reuse between DSP blocks introduces large fan-out[44]. Moreover, the timing closure get worse on Xilinx ultrascale+ VU9P FPGA which enables multiple super-logic regions (SLR) design. In this case, signal paths between two SLR may introduce large delays and significantly impact timing. DarkFPGA tackles the timing issue for two major optimization. First, a multiple level of pipelining registers is inserted after high fan-out data stream as a balanced tree to limits the fan-out it can have. Second, the GEMM Kernel of DarkFPGA is forced to be divided into two parallel submodules, which are assigned to a super-logic regions (SLR). These signal interconnects of GEMM Kernel are limited within their cor-

Table 2. The network architecture in experiment.

Layer	B	C	F	$H \times W$	K
CONV1	128	3	128	32×32	3×3
CONV2	128	128	128	32×32	3×3
MAXPOOLING	128	128	128	16×16	2×2
CONV3	128	128	256	16×16	3×3
CONV4	128	256	256	16×16	3×3
MAXPOOLING	128	256	256	8×8	2×2
CONV5	128	256	512	8×8	3×3
CONV6	128	512	512	8×8	3×3
MAXPOOLING	128	512	512	4×4	2×2
FC	128	8096	1024	–	–
FC	128	1024	10	–	–
SSE	128	10	10	–	–

responding super-logic regions (SLR).

8.1. Exploration of DarkFPGA performance

Based on the discussions in Section 5 and Section 6, the performance of DarkFPGA is significantly determined by the tile sizes (T_B, T_I). Therefore, the selection of tile parameters is analyzed here under the network configuration shown in Table 2.

The batch tile T_B is crucial for maximising performance, which is bounded by the training batch size. Consider that the commonly used batch size is 128, the selection of batch tile for exploration T_B is constrained among (32, 64, 128). Also, the image tile T_I is mainly related to the computational times T_{CONV} and T_{FC}, which depends on the network parameters C, F, H and W. According to Table 2, the selection of image tile T_I is constrained among (16, 32, 64). Due to the requirements of matrix transposition operations, the batch tile T_B should be greater than or equal to image tile T_I. Therefore the design space (T_B, T_I) under evaluation is constrained within (128, 64), (128, 32), (128, 16), (64, 64), (64, 32), (64, 16), (32, 32), (32, 16), (16, 16). In particular, the design space (128, 64) is removed from evaluation for hardware resource limitation.

The corresponding performance and resource consumption under different design parameters (T_B, T_I) is illustrated in Fig. 8. As shown in Figs. 8(a) and 8(b), the design space (T_B, T_I) changes from (128, 32) to (32, 16) as the performance decreases. We can find that the most critical factors that affect the performance are the multiplication of two tiling size ($T_B \times T_I$). The reason is that, according to our performance model, the computational time for matrix multiplication operations are accelerated by ($T_B \times T_I$) times, which domain the training process. And the secondary influencing factor is batch-level parallelism (T_B). This is because the auxiliary operations are only accelerated by T_B times, with no relationship about T_I. Finally, Fig. 8(c) shows the relation between DSP utilization and performance, indicating that our design space is bounded by DSP resources.

Therefore, we customize a DarkFPGA design to determine the optimal implementation of the training accelerator when $T_B = 128$, $T_I = 32$.

8.2. Heterogeneous versus homogeneous computing

Some of the existing FPGA accelerators rely on heterogeneous computing to handle auxiliary operations[36, 45]. To quantitatively compare the performance discrepancies between heterogeneous and homogeneous computing, our DarkFPGA framework is revised to implement heterogeneous computation across an FPGA-CPU heterogeneous system, which is achieved by delivering auxiliary operations to CPU and removing the auxiliary kernels.

In this experiment, the tiling size is set to ($T_B = 128, T_I = 32$). Fig. 9 illustrate the experiment results which show that homogeneous computing can achieve significantly higher performance on our DarkFPGA framework. Based on the comparison between CPU homogeneous system and CPU+FPGA heterogeneous system (Fig. 9(a) versus Fig. 9(b)), heterogeneous computing can effectively improve the performance of GEMM, but other parts of the training would become a new computing bottleneck. This problem can be addressed with a homogeneous FPGA system, accelerating everything by batch-parallelism to achieve over 10 times speedup (Fig. 9(b) versus Fig. 9(c)). This experiment clearly showcases the benefits of implementing the entire training process on the FPGA.

Note that using multi-threaded or high-performance CPU can significantly improve heterogeneous computing performance. However their high power consumption brings a tough challenge for embedded DNN applications.

8.3. Performance comparison with GPU and CPU

Here we compare the performance of DarkFPGA with other platforms like GPU and CPU. All software results are running on an Intel Xeon X5690 CPU (6 cores, 3.47 GHz) and an NVIDIA GeForce GTX 1080 Ti GPU. After finishing the same number of batches, all platforms achieve similar accuracies. Unfortunately, GeForce GTX 1080 Ti does not have native int8 support, we evaluate the GPU performance by limiting the range of float32 number system, instead of actual GPU low-precision training.

Table 3 shows the performance and power consumption, as well as other important metric on different platforms. DarkFPGA can achieve over 200 times speedup over a CPU-based implementation of Darknet and is 2 times slower than a GPU-based implementation of Darknet on overall performance. The average power consumption (13.5 W) of FPGA is obtained by the Maxeler performance monitoring tools. By multiplying time and power consumption, our FPGA-based design is 5 times more energy efficient than GPU implementation of Darknet.

Note that Darknet is a lightweight neural network framework for fast iterative design, which limits the overall performance of GPU and CPU. For fair comparison, we evaluate the training performance on TensorFlow[46] using multi-threaded acceleration for CPU and cuDNN[47] acceleration for GPU, as shown in brackets. It shows that DarkFPGA achieves 10 times speed up over CPU-based implementation and 2.5 times more energy efficient than GPU implementation on TensorFlow.

8.4. Performance comparison with other FPGA-based training system

Finally, comparisons between out DarkFPGA and other FPGA training accelerators are conducted, in terms of resource utilization, performance and throughput and on Table 4. Since many important matrices are not provided and the models for training are different, comparison between different FPGA-based training accelerator in a fair is extremely difficult

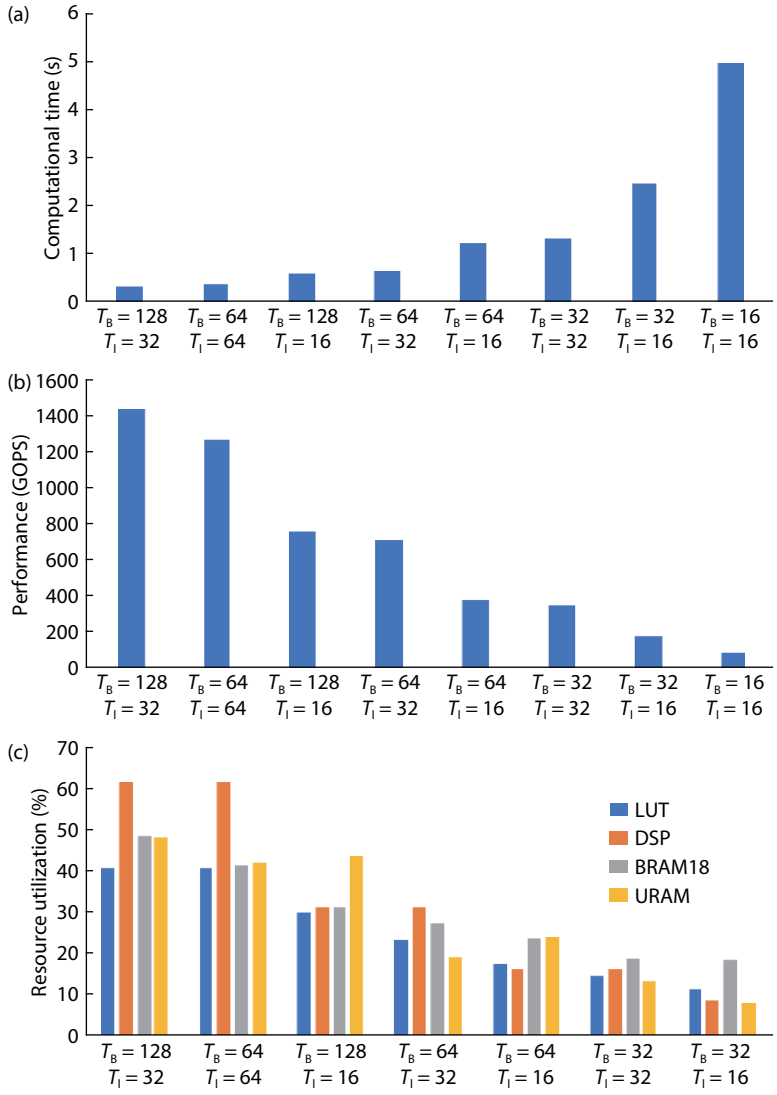


Fig. 8. (Color online) Performance and resource consumption experiments under different design space using int8 weights. (a) Computational time. (b) Performance evaluation. (c) Resource consumption.

Table 3. Performance comparison among FPGA, CPU and GPU.

Parameter	CPU	GPU	DarkFPGA
Platform	Intel Xeon X5690	GTX 1080 Ti	MAX5 Platform
No. of cores	6	3584	–
Compiler	GCC 5.4.0	CUDA 9.0	Maxcompiler 2019.2
Flag	-Ofast	–	–
Frequency (GHz)	3.47	1.58	0.2
Precision	32-bit floating point	32-bit floating point	8-bit fixed point
Technology (nm)	32	28	16
Processing time per batch (ms)	66439 (3270)	126 (53.4)	331
Threads	1 (24)	–	–
Power (W)	131 (204)	187 (217)	13.5
Energy (J)	8712 (667.1)	23.6 (11.6)	4.5
Energy efficiency	1x (13x)	369x (751x)	1936x

and not straightforward. Even in such situation, our DarkFPGA accelerator still show desirable performance. Our design outperforms FPDeep[31] in term of performance per FPGA. Accelerator from DiCecco *et al.*[33] outperform our accelerator in throughput by training small network similar to LeNet-5[1]. Moreover, in comparison to[35], a lightweight FPGA training accelerator, our DarkFPGA maintains a high throughput when training more smaller network VGG-16[43] compared with[35]. A work from[34] achieves much higher throughput than our implementation by pruning over 90% weights and 50% activation during training process, whose performance can be 4× faster than GPU.

These comparisons can somehow show that our DarkFPGA has the capacity to train deep neural network. Our analyses show that the improvement of performance comes from FPGA-based batch-level parallelism. In particular, the dataflow architecture allows us to fully exploit the advantages of batch-level parallelism and maximize throughput with the help of dataflow programming language[19].

9. Conclusion

This work proposes DarkFPGA, a novel FPGA framework for efficient training of deep neural networks, with a customized low-precision DNN training algorithm. The DarkFPGA accelerator explores batch-level parallelism, which provides efficient training acceleration for both forward and backw-

Table 4. Performance comparison of different FPGA-based training accelerators.

Accelerator	Platform	Config	Model dataset	LUTs (kW)	DSPs efficiency	Performance (GOPS)	Throughput (image/s)
F-CNN[29] FCCM 16	Altera Stratix V	8 FPGA	LeNet-5 MNIST	–	–	–	7
FPDeep[31] FCCM 18	Virtex7 VC709	10 FPGA	AlexNet imagenet	≈ 460 per FPGA	≈ 2880 per FPGA	≈ 1022 per FPGA	–
DiCecco *et al.*[33] FPGA 18	Xilinx XCKU115	1 FPGA	LeNet-like CIFAR10	≈ 530	≈ 883	–	522
Nakahara *et al.*[34] FPL 19	UltraScale+ XCVU9P	1 FPGA	VGG16 CIFAR10	934	1106	–	4878
Sean *et al.*[35] FPT 19	Zynq ZCU111	1 FPGA	VGG-16 CIFAR10	73.1	1037	–	3.3
DarkFPGA	UltraScale+ XCVU9P	1 FPGA	VGG-like CIFAR10	480	4202	1417	386.7

(1) '–' means this metrics is not provided on their papers, '≈' indicate that this value is obtained by approximate estimates. (2) The accelerator from Ref. [29] didn't compute the gradients for training. (3) The power consumption of Ref. [29] measured from entire development board when our power consumption is measured from single FPGA chip.

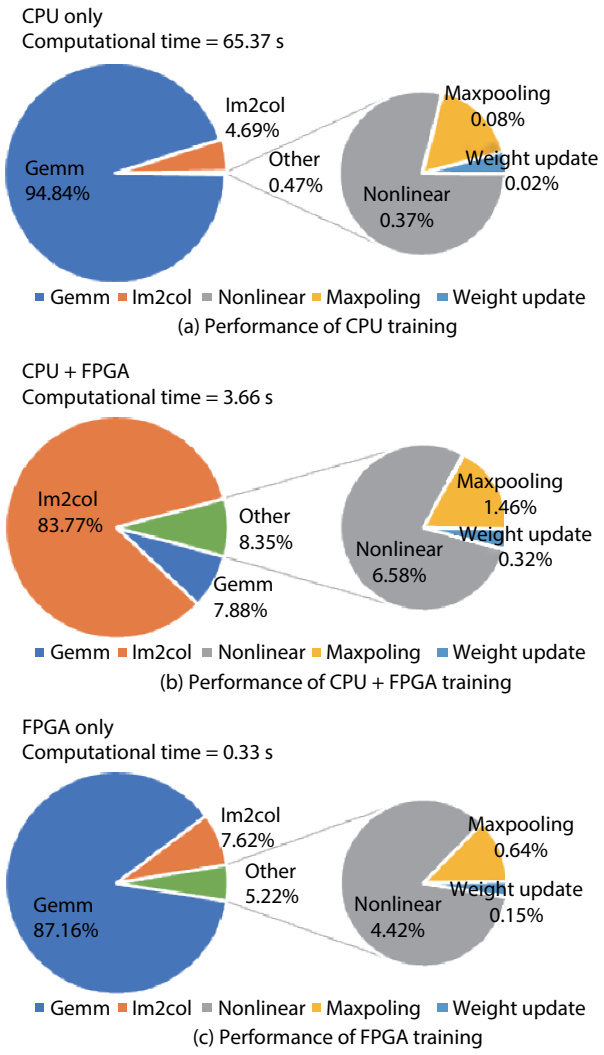


Fig. 9. (Color online) Performance comparisons between homogeneous system and heterogeneous system.

ard propagation on a homogeneous FPGA system. Optimization strategies such as batch-focused data sequence CHWB and tiling strategies are employed to improve overall performance. Furthermore, an optimization tool is developed for determining the optimal design parameters for a specific network description. Future work includes applying DarkFPGA to multi-FPGA clusters, exploring mixed precision and binarised training, and supporting cutting-edge network functions like group normalization and depthwise convolution.

References

[1] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition. Proc IEEE, 1998
[2] Russakovsky O, Deng J, Su H, et al. Imagenet large scale visual recognition challenge. IJCV, 2015
[3] Ren S, He K, Girshick R, et al. Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in Neural Information Processing Systems, 2015, 91
[4] He K, Gkioxari G, Dollár P, et al. Mask r-cnn. Proceedings of the IEEE International Conference on Computer Vision, 2017, 2961
[5] Jia Y, Learning semantic image representations at a large scale. PhD Thesis, UC Berkeley, 2014
[6] Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015
[7] Umuroglu Y, Fraser N J, Gambardella G, et al. Finn: A framework for fast, scalable binarized neural network inference. Acm/sigda International Symposium on Field-Programmable Gate Arrays, 2016
[8] Nurvitadhi E, Venkatesh G, Sim J, et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017
[9] Guo K, Zeng S, Yu J, et al. A survey of FPGA-based neural network accelerator. arXiv: 171208934, 2017
[10] Parisi G I, Kemker R, Part J L, et al. Continual lifelong learning with neural networks: A review. arXiv: 180207569, 2018
[11] Micikevicius P, Narang S, Alben J, et al. Mixed precision training. arXiv: 171003740, 2017
[12] Das D, Mellempudi N, Mudigere D, et al. Mixed precision training of convolutional neural networks using integer operations. arXiv: 180200930, 2018
[13] Banner R, Hubara I, Hoffer E, et al. Scalable methods for 8-bit training of neural networks. arXiv: 180511046, 2018
[14] De Sa C, Leszczynski M, Zhang J, et al. High-accuracy low-precision training. arXiv: 180303383, 2018
[15] Wu S, Li G, Chen F, et al. Training and inference with integers in deep neural networks. arXiv: 180204680, 2018
[16] Wen W, Xu C, Yan F, et al. Terngrad: Ternary gradients to reduce communication in distributed deep learning. Advances in Neural Information Processing Systems, 2017
[17] Zhu H, Akrout M, Zheng B, et al. Benchmarking and analyzing deep neural network training. IEEE International Symposium on Workload Characterization (IISWC), 2018
[18] Redmon J. Darknet: Open source neural networks in C. http://pjred-

die.com/darknet/

[19] Pell O, Mencer O, Tsoi K H, et al. Maximum performance computing with dataflow engines. High-performance computing using FPGAs, 2013

[20] Luo C, Sit M K, Fan H, et al. Towards efficient deep neural network training by FPGA-based batch-level parallelism. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, 45

[21] Kingma D P, Ba J. Adam: A method for stochastic optimization. arXiv: 14126980, 2014

[22] Qiu J, Wang J, Yao S, et al. Going deeper with embedded FPGA platform for convolutional neural network. Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016

[23] Suda N, Chandra V, Dasika G, et al. Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks. Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016

[24] Motamedi M, Gysel P, Akella V, et al. Design space exploration of FPGA-based deep convolutional neural networks. ASP-DAC, 2016

[25] Zhang C, Sun G, Fang Z, et al. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. IEEE Trans Comput-Aid Des Integr Circuits Syst, 2019, 38, 2072

[26] Ma Y, Cao Y, Vrudhula S, et al. An automatic rtl compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. 2017 27th International Conference on Field Programmable Logic and Applications (FPL), 2017, 1

[27] Venkataramanaiah S K, Ma Y, Yin S, et al. Automatic compiler based FPGA accelerator for cnn training. 2019 29th International Conference on Field Programmable Logic and Applications (FPL), 2019, 166

[28] Xiao Q, Liang Y, Lu L, et al. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. Proceedings of the 54th Annual Design Automation Conference, 2017

[29] Zhao W, Fu H, Luk W, et al. F-CNN: An FPGA-based framework for training convolutional neural networks. IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016

[30] Geng T, Wang T, Li A, et al. A scalable framework for acceleration of cnn training on deeply-pipelined FPGA clusters with weight and workload balancing. arXiv: 190101007, 2019

[31] Geng T, Wang T, Sanaullah A, et al. Fpdeep: Acceleration and load balancing of CNN training on FPGA clusters. IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018

[32] Li Y, Pedram A, Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks. IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2017

[33] Dicecco R, Sun L, Chow P. FPGA-based training of convolutional neural networks with a reduced precision floating-point library. International Conference on Field Programmable Technology, 2018

[34] Nakahara H, Sada Y, Shimoda M, et al. FPGA-based training accelerator utilizing sparseness of convolutional neural network. 2019 29th International Conference on Field Programmable Logic and Applications (FPL), 2019, 180

[35] Fox S, Faraone J, Boland D, et al. Training deep neural networks in low-precision with high accuracy using FPGAs. International Conference on Field-Programmable Technology (FPT), 2019

[36] Moss D J, Krishnan S, Nurvitadhi E, et al. A customizable matrix multiplication framework for the Intel HARPv2 Xeon+ FPGA platform: A deep learning case study. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018, 107

[37] He K, Zhang X, Ren S, et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Proceedings of the IEEE International Conference on Computer Vision, 2015, 1026

[38] Zhou S, Wu Y, Ni Z, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv: 160606160, 2016

[39] Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul, 1998, 8(1), 3

[40] Performance guide of using nchw image data format. [Online]. Available: https://www.tensorflow.org/guide/performance/overview

[41] Ma Y, Cao Y, Vrudhula S, et al. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017

[42] Steinkraus D, Buck I, Simard P. Using GPUs for machine learning algorithms. Eighth International Conference on Document Analysis and Recognition (ICDAR), 2005

[43] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv: 14091556, 2014

[44] Wei X, Yu C H, Zhang P, et al. Automated systolic array architecture synthesis for high throughput cnn inference on FPGAs. Proceedings of the 54th Annual Design Automation Conference, 2017,

[45] Krishnan S, Ratusziak P, Johnson C, et al. Accelerator templates and runtime support for variable precision CNN. CISC Workshop, 2017

[46] Abadi M, Barham P, Chen J, et al. TensorFlow: a system for large-scale machine learning. OSDI, 2016, 265

[47] Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: Efficient primitives for deep learning. arXiv: 14100759, 2014