2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Towards Efficient Deep Neural Network Training by FPGA-based Batch-level Parallelism

Cheng Luo *†[1], Man-Kit Sit†, Hongxiang Fan†, Shuanglong Liu†,Wayne Luk† and Ce Guo†
*State Key Laboratory of ASIC and System, Fudan University, Shanghai, China
Email:*cluo16@fudan.edu.cn
†Department of Computing, Imperial College London, London, United Kingdom
Email: {m.sit18, h.fan17, s.liu13, w.luk, c.guo}@imperial.ac.uk

***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 training on FPGAs. 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 11 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.**

I. Introduction

Deep neural networks (DNNs) have achieved remarkable results on various demanding applications such as image classification [1] and object detection [2]. In resource-limited settings, the development of real-time and low-power hardware accelerators is especially critical, and hence various hardware devices such as 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 DNN algorithms [3].

However, most of the existing FPGA accelerators are designed for inference with low-precision DNN models, which are pre-trained on high-precision models (e.g. 32/64-bit floating point models). Since DNNs employ different precision formats for training and inference, they often need further fine-tuning to achieve acceptable accuracy. This makes them diffcult to support, for example, systems requiring continual learning [4]. Various low-precision training techniques, such as mixed precision [5], fixed-point [6, 7] and ternary [8] weight parameters, have been proposed to reduce the overhead associated with fine-tuning to create low-precision models.

In this paper, we explore the benefits and drawbacks of employing FPGA and GPU platforms for low-precision training. In particular, we develop an FPGA framework that supports DNN training on a single FPGA with a low-precision number system using 8-bit integer (`int8`). Our objective is to determine if the fine-grained customizability and flexibility offered by FPGAs can be exploited to outperform state-of-the-art GPUs in low precision training in terms of speed and power consumption.

To meet this objective, we need to address the following challenges.

1) Compared to DNN inference, the training process requires more memory and computational resources of FPGAs due to the additional computations and different operations performed in backward propagation [9]. This leads to differences in requirements for hardware architecture.
2) FPGA implementations of DNN inference usually exploits image-level and layer-level parallelism to reduce latency [10]. However, training usually proceeds with batches of training examples in parallel. Effective exploitation of such batch-level parallelism can be the key to significant acceleration.

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.

Moreover, we propose DarkFPGA, an FPGA-based deep learning framework with a dataflow architecture. Our approach is built on Darknet framework [11], an open-source neural network framework written in C and CUDA, with FPGA implementation using MaxJ [12]. To the best of our knowledge, it is the first training framework on FPGA supporting low-precision DNN training.

In summary, this paper makes the following contributions.

- A novel accelerator for a complete DNN training process. A dataflow architecture that explores batch-level parallelism for efficient FPGA acceleration of DNN

[1]This work is done when Cheng Luo was visitting Imperial College London

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00016

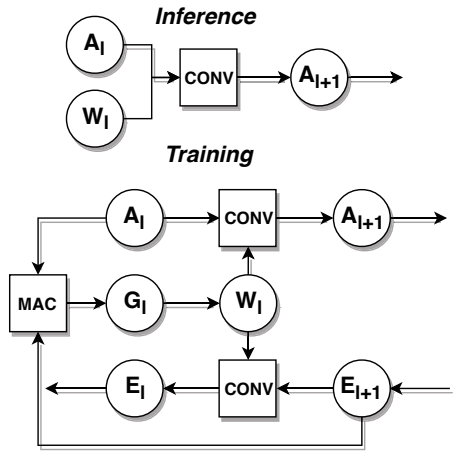


Figure 1. A comparison of the convolutional layer for inference and training.

training is developed, providing a power-efficient and high-performance solution for efficient training.

- 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 in ternary weights.
- An automatic optimization tool for the framework to explore the design space to determine the optimal parameters for a given network specification.

II. Background

This section provides an overview of DNN training, emphasizing its difference from inference, and then presents state-of-the-art FPGA implementations of DNN acceleration.

A. Training versus Inference

The major difference between training and inference is that the training process requires an additional process of backward propagation to compute the gradients of the cost function, subsequently using them to update the model weights so that the model can show desirable behavior. Backward propagation makes training computationally expensive and requires different operations in comparison to inference.

Figure 1 shows the operations needed for the inference and training of a convolutional layer. For a layer l, the inference process consists of forward propagation, which 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 the contrary, the training process first performs forward propagation as during inference to compute the errors using the loss function. Thereafter, backward propagation is performed 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), which is also used to compute the gradients (G_l) with respect to the loss function. The gradients are used to update the current weights according to the chosen optimization algorithm. Algorithm 1 presents the pseudocode for the training of a convolutional layer to provide a precise description for the process, The meaning of the notations can be found in Table I, where the same set of notation is also followed in the rest of this paper.

Algorithm 1: Pseudocode for Training Convolutional layers

```
1  Forward:
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:
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]
```

Therefore, the required computations for training is almost three times of that for inference as the latter requires only executing the **Forward** loop. In addition, it requires significantly more memory space for storing errors and gradients.

B. Related Work

Most FPGA implementation efforts mainly focus on the acceleration of DNN inference. One notable work [10] exploits image-level and layer-level parallelism extensively to achieve state-of-the-art speedup for inference. For training [13, 14, 15, 16], Geng et al. [13] explores layer-level parallelism for training a model on multiple FPGAs in a pipelined manner. Moreover, Li et al. [14] studies reconfigurable communication patterns when training on a multi-FPGA platform. However, the performance will be undesirable if we naively deploy the inference architecture naively for training without considering the enormous number of training examples and the extra backward operations required for training [17].

Some FPGA implementations [16, 18] attempt to tackle the problem by distributing the computations across a heterogeneous FPGA-CPU system. Moss et al. [18] proposes to perform the core GEMM operations on FPGAs and leave CPU for the remaining jobs. However, this solution requires effective load balancing support for heterogeneous devices, since unpredictable communication cost between CPUs and FPGAs can make the communication bound operations a new bottleneck of the design. Based on our profiling in Section VI, the operations executed on CPU 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. To the best of our knowledge, our work is the first low-precision DNN training framework accelerated on a single FPGA platform. Compared

TABLE I
PARAMETERS FOR FPGA IMPLEMENTATIONS OF TRAINING.

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

to other frameworks, our proposed customizable FPGA design achieves 11 times speedup over a CPU-based implementation and is about 3 times more energy efficient than a GPU-based implementation.

III. FPGA ACCELERATORS FOR DNN TRAINING

In this section, we present the use of the CHWB pattern to explore the batch-level parallelism for training, and show how we can apply this idea to develop the architecture of our training accelerator.

A. CHWB Pattern

For training, the activations, weights, errors and gradients are too large to be stored completely in the on-chip memory on an FPGA. Therefore, 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, it is necessary to explore an optimal data access pattern to efficiently utilize the bandwidth, particularly for training.

In this work, the most widely-used data pattern for DNN training on GPUs is referred as Batch-Channel-Height-Width (**BCHW**), which depicts the order of data dimensions in the memory space [19], where the elements along the lowest dimension **W** are stored consecutively. Figure 2(a) shows an example of data represented in the BCHW pattern along with its corresponding data layout in the DRAM in Figure 2(c). In this case, it is difficult to fetch the elements from different batches in burst mode since they are usually not stored consecutively in memory, which under-utilize the bandwidth when exploring batch-level parallelism.

We develop the Channel-Height-Width-Batch (**CHWB**) pattern for FPGAs to explore batch-level parallelism without compromising bandwidth utilization. As shown in Figure 2(b), the elements from adjacent batches are allocated consecutively, which allows the memory interface to simultaneously read multiple training examples. As a result, our accelerator needs only a single DRAM burst access to acquire all necessary input data which greatly improves bandwidth utilization.

B. Tiling

Tiling is a common optimization technique to improve memory bandwidth and computing resource utilization for DNN acceleration on resource-limited FPGA devices. The strategy partitions large input frames into smaller tiles of data so that each tile can be fitted into the on-chip memory of an FPGA. It is particularly important for training as some resource-intensive tasks like matrix transpose are needed during backward propagation.

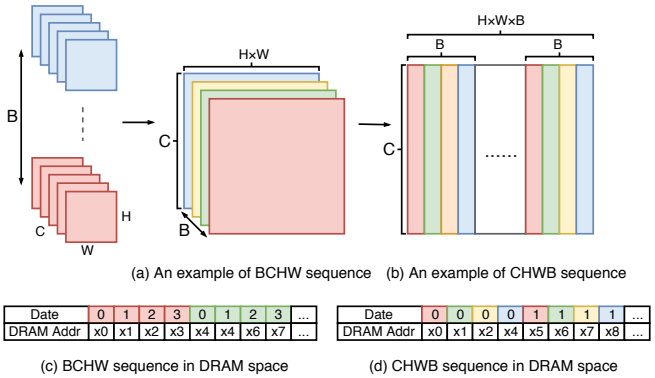


Figure 2. Comparison of two sequences of BCHW and CHWB.

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 [20], we set the image tile T_I equal to channel tile T_C and filter tile T_F. Therefore, we explore two levels of parallelism 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.

The tiling parameters must be chosen carefully in order to maximize the performance of the design. A larger tile size reduces the number of data transfers and increases parallelism, but it requires more on-chip memory for storing tiles. In Section IV. we will explore the optimization of the tile sizes.

C. System Overview

Figure 4 presents an overview of our FPGA-based training accelerator, which consists of a computation kernel, a global controller and a DDR controller for off-chip memory transfer. The computational kernel has a batch splitter, a set of processing elements (PEs) and a batch merger. When a stream of training batches is arriving at the kernel from the DDR controller, 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 it is 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.

D. Unified GEMM Kernel

Figure 5 shows the architecture of the GEMM kernel. The kernel provides a unified datapath to support the convolutional

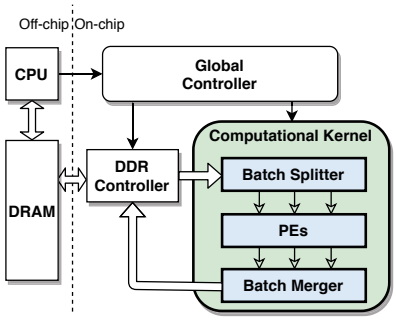


Figure 4. System overview.

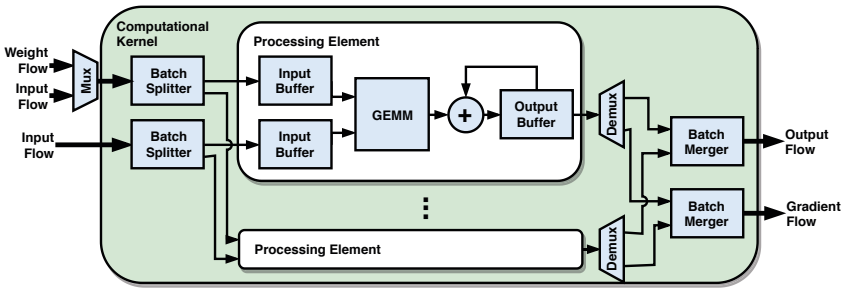


Figure 5. Hardware architecture of GEMM kernel.

and fully-connected computations of the forward and backward propagation, as well as the gradient generation, which are the most computationally intensive tasks in the accelerator. This unified approach is adopted because matrix multiplication is the common operation for these computations and only the input/output matrix to/from the kernel needs to be changed. Therefore, we can avoid time-consuming dynamic reconfiguration [16] or using separate kernels for different operations [13].

Before any computation, the data streams are stored in the input buffers, which are organized as a double buffer in order to overlap the data transfer with the computation. When the $(i+1)^{th}$ tile flows from the batch splitter to the input buffer, the i^{th} tile can be sent to the GEMM kernel for the computation. Note that for the weight stream, as the weight data are shared by different tiles across the same batch, the weight stream bypasses the batch splitter and enters the input buffers directly.

The GEMM kernel fetches data from the input buffers and performs tiled matrix multiplication. The intermediate values during each iteration are stored in the output buffers and are used for the next iteration. The final results are post-processed by the batch merger for forward/backward propagation and gradient computation, then transferred back to the DRAM. The details of the tiled matrix multiplication for the convolutional and fully-connected layers are shown in Algorithm 2.

In order to support different modes of operations in a single datapath, the global controller dynamically re-configures the buffers and data flow to set up the datapath to operate differently. The input buffer can be configured to perform on-the-fly matrix transposition for the computation of backward propagation. Furthermore, the global controller switches the multiplexer to feed the correct input streams to the processing elements and the demultiplexer to direct the output stream to the appropriate postprocessing unit.

E. Auxiliary Kernels

The auxiliary kernels are designed for accelerating supplementary operations with batch-level parallelism. The operations include *im2col, col2im, max-pooling, reshape, weight updating, summation and nonlinear functions.* as well as their backward counterparts (if necessary). Unlike the operations in the convolutional or fully-connected layers, these operations have no learnable weights and require only a small amount of computation.

Algorithm 2: Pseudocode of Tiled Matrix Multiplication for Convolutional and Fully-connected Layers

Consider input frames, output frames and error frames as 3-dimensions $T_B \times T_I \times T_I$ tiled blocks, Weight matrix and gradient matrix is also transferred into $T_I \times T_I$ tiled blocks.
Convolutional layer:
for f = *1 to* F/T_I **do**
  for im = *1 to* $(H*W)/T_I$ **do**
    for b = *1 to* B/T_B **do**
      for c = *1 to* $C \times K/T_I$ **do**
        $A_{l+1}(b)(f,im)_c = W_l(f,c) \times A_l(b)(c,im)$
        $A_{l+1}(b)(f,im) \mathrel{+}= A_{l+1}(b)(f,im)_c$
      Output $A_{l+1}(b)(f,im)$
Convolutional gradients generations:
for f = *1 to* F/T_I **do**
  for c = *1 to* $C \times K/T_I$ **do**
    for b = *1 to* B/T_B **do**
      for im = *1 to* $(H*W)/T_I$ **do**
        $G_l(b)(f,c)_{im} = E_{l+1}(b)(f,im) \times A_l(b)(c,im)^T$
        $G_l(b)(f,c) \mathrel{+}= G_l(b)(f,c)_{im}$
      Output $G_l(b)(f,c)$
Fully-connected layer:
for f = *1 to* F/T_I **do**
  for b = *1 to* B/T_B **do**
    for c = *1 to* C/T_I **do**
      $A_{l+1}(b)(f)_c = A_l(b)(c) \times W_l(f,c)^T$
      $A_{l+1}(b)(f) \mathrel{+}= A_{l+1}(b)(f)_c$
    Output $A_{l+1}(b)(f)$

The kernels consist of various types of separate processing units to support the operations, receiving the parallel data streams from the batch splitter and send them to the specific units. These units store the input stream in a line buffer, and a window captures part of the pixels in the buffer to output, compare or accumulate. Finally, the output streams flow through the batch merger as a batch stream for the DRAM.

IV. 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.

A. 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 Fre are:

$$BW_{CONV} = (T_B \times L_I + T_B \times T_I/N \times L_O + L_W) \times Fre$$
$$BW_{FC} = (T_B \times L_I + T_B \times T_I/N \times L_O + T_I \times L_W) \times Fre$$

where L_I is the bitwidth of the input activations, L_O is the bitwidth of the output activations and L_W is the bitwidth of the weights.

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:

$$BW_{auxiliary} = (2 \times T_B \times L_I + 2 \times T_B \times L_O) \times Fre$$

B. 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. We therefore define them as LUT_{fix}, DSP_{fix}, $BRAM_{fix}$.

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:

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

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

$$DSP = T_B \times T_I \times D_{mul} + T_B \times A_l \times D_{add} + T_B \times D_{add}$$

where D_{mul} and D_{add} are the DSP usage of the multiplier and adder affected by the bitwidth of activations and weights. A_l 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 consumption of LUT as it is difficult to predict statically. Its usage is given by:

$$LUT = T_B \times T_I \times \beta + T_B \times \delta$$

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

C. Performance Modeling

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

$$T_{CONV} = \frac{B \times C \times K \times F \times H \times W}{T_I \times T_B \times Fre}$$
$$T_{FC} = \frac{B \times C \times F}{T_I \times T_B \times Fre}$$

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. Therefore, the actual computational time should be:

$$T_{CONV} = \frac{\lceil B \rceil^{T_B} \times \lceil C \times K \rceil^{T_I} \times \lceil F \rceil^{T_I} \times \lceil H \times W \rceil^{T_I}}{T_B \times T_I \times Fre}$$
$$T_{FC} = \frac{\lceil B \rceil^{T_B} \times \lceil C \rceil^{T_I} \times \lceil F \rceil^{T_I}}{T_B \times T_I \times Fre}$$
$$\lceil X \rceil^T = ceil(X/T) * T$$

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.

Also, in each cycle of the auxiliary kernel, frames batches can be handled simultaneously and the processing time is:

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

For our dataflow architecture, the transmission time of the computational kernels is overlapped by the computation time. In this manner, the communication time should meet the bandwidth requirement.

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

$$Minimize\ Time = T'_{CONV} + T'_{FC} + T'_{auxiliary}$$
$$where \begin{cases} LUT + LUT_{fix} \leq LUT_{limit} \\ BRAM + BRAM_{fix} \leq BRAM_{limit} \\ DSP + DSP_{fix} \leq DSP_{limit} \\ BW \leq BW_{limit} \end{cases}$$

where LUT_{limit} , $BRAM_{limit}$, DSP_{limit} , BW_{limit} are limited FPGA resources, and T'_{CONV} + T'_{FC} + $T'_{auxiliary}$ are the expected computation time for a specific network description.

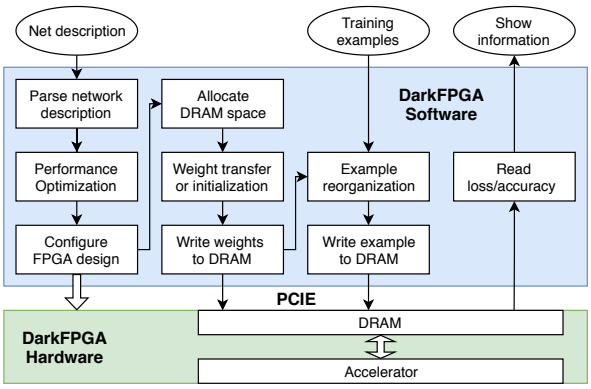


Figure 5. The DarkFPGA framework.

V. The DarkFPGA Framework

For our proposed dataflow architecture, we present DarkFPGA, a hardware/software co-designed FPGA framework for effective training. Overall, the key features of DarkFPGA include:

1) Scalable accelerator architecture. We develop a multi-level parallelism scalable FPGA design that can easily be scaled up to larger devices with more resources.
2) Software definable hardware accelerator. Our accelerator can be configured by software to support various DNN networks and different parallel levels through deploying different FPGA bitstreams.
3) A tool for optimizing hardware design. Our tool accepts a set of user constraints including network description and hardware resource description to produce optimised performance.

We automate the process of exploring design parameters for the DarkFPGA framework. The users only need to provide a network description and a training dataset, then it will produce the most suitable parameters for the accelerator for the given network. DarkFPGA hardware accelerates the entire training process with a unified module on FPGA. Our tool, illustrated in Figure 5, has 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. Note that the weights are transferred during initialisation and are updated subsequently on FPGA.
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. When the FPGA is engaged in entire training, no computational operations are required by the host CPU.
6) Train neural network iteratively. Transfer loss and accuracy information back to the host for each complete training batch.

Table II
The network architecture in experiment

Layer	B	C	F	H × 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	-	-
SSE	128	10	10	-	-

VI. Experimental Result

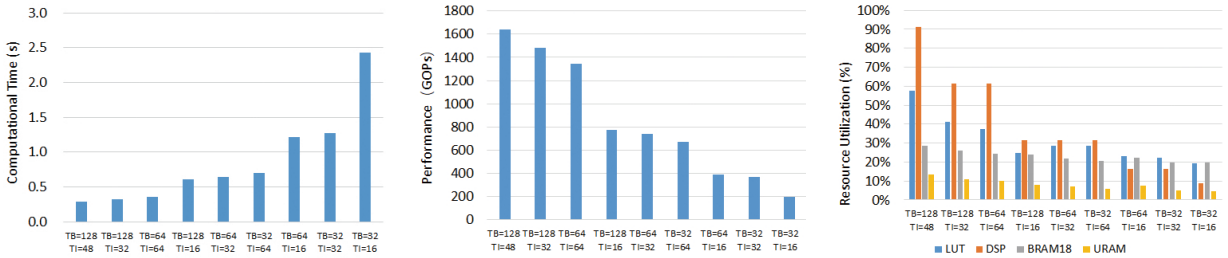
To find the performance and limitation of the DarkFPGA framework, we evaluate our framework on the Maxeler MAX5 platform, which consists of a Xilinx ultrascale+ VU9P FPGA. Three 16GB DDR4 DIMMs are installed on the platform as off-chip memory with a maximum bandwidth of 63.9GB/s. Our hardware accelerator works at 200 MHz. Maxcompiler 2018.2 and Vivado 2017.2 are used for synthesis and implementation. The VGG-like network [21] trained on the Cifar10 [22] dataset is evaluated in the following training experiments, which achieves 93% Top-5 accuracy on our DarkFPGA system. Shuang et. al [8] shows that this network can be trained with 8-bit integers and the sum-square-error loss function (SSE).

A. Exploration of DarkFPGA performance

Based on the discussions in Section III and Section IV, the performance of DarkFPGA for a specific network description is determined by the tile sizes (T_B, T_I). We analyze the choice of tile sizes for the network configuration shown in Table II.

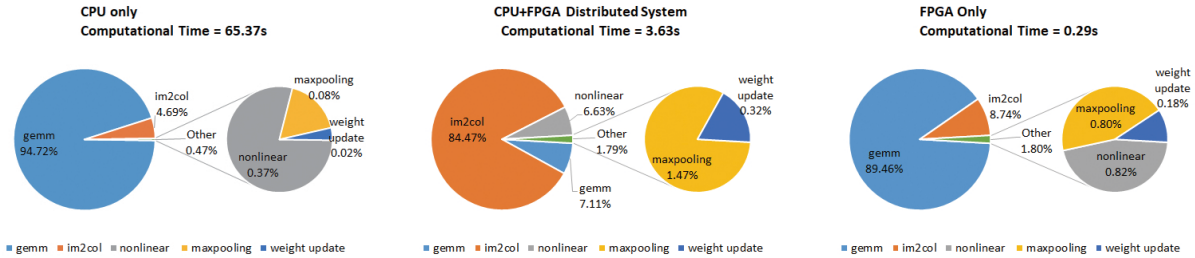
The batch tile T_B is crucial for maximising performance and is bounded by the training batch size. Consider that the commonly used batch size is 128, we select the three factors of 128 as (32,64,128) to set batch tile size T_B for exploration. The image tile T_I is mainly related to the computational times T_{CONV} and T_{FC}, which depends on the network parameters C, F, H and W. Based on our observation from Table II, we choose (16, 32, 64) for T_I to explore the design space. Additionally, the maximal supported design parameters in our framework $(T_B, T_I) = (128, 48)$ are also evaluated.

Figure 6 shows the corresponding performance and resource consumption under different design parameters (T_B, T_I). As shown in Figure 6(a, b), the design space (T_B, T_I) changes from (128,48) to (32,16) as the performance decreases. We find that the most critical factors that affect the performance are the multiplication of two tiling size($T_B \times T_I$) and batch-level parallelism (T_B). 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, while the auxiliary operations are accelerated by T_B times.



(a) Computational time of training (b) Performance of training (c) Resource consumption

Figure 6. Performance and Resource consumption experiments under different design space



(a) Performance of CPU training (b) Performance of CPU+FPGA training (c) Performance of FPGA training

Figure 7. Performance comparisons between homogeneous system and heterogeneous system

As the matrix multiplication operations dominate the training process, the number of multiplications ($T_B \times T_I$) determines the overall performance. Also at the same level of $T_B \times T_I$, larger batch-level parallelism T_B leads to slightly better performance. Finally, Figure 6(c) shows the relation between DSP utilization and performance, indicating that our design space is bounded by DSP resources.

Therefore, we can customize a DarkFPGA design to determine the optimal implementation of the training accelerator when $T_B = 128$, $T_I = 48$.

B. *Heterogeneous versus Homogeneous Computing*

Some of the existing FPGA accelerators rely on heterogeneous computing to handle auxiliary operations on training and inference [18, 23]. 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 ($T_B = 128, T_I = 48$). The result is presented in Figure 7 which shows that homogeneous computing can achieve significantly higher performance. Based on the comparison between CPU homogeneous system and CPU+FPGA heterogeneous system (Figure 7(a) versus Figure 7(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 (Figure 7(b) versus Figure 7(c)). This 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.

C. *Performance Comparison with GPU and CPU*

To compare the performance of DarkFPGA with other platforms, the final implementation is customized for performance with int8. All software results are running on an Intel Xeon X5690 CPU (6 cores, 3.47GHz) 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, so the performance of GPU low-precision training is evaluated by limiting the range of float32 number system. Comparison with GPUs supporting int8 is planned for future work.

Table IV shows the performance and power consumption on different platforms. DarkFPGA achieves over 200 times speedup over a CPU-based implementation of Darknet and is 2.5 times slower than a GPU-based implementation of Darknet on overall performance. Since we use a homogeneous FPGA platform, we compare the power consumption of FPGA and GPU accelerators. By multiplying time and power consumption, we can see that our FPGA-based design is 6.5 times more energy efficient than GPU implementation of Darknet.

Note that Darknet is a lightweight neural network framework, which allows us to develop FPGA training framework easily but also limits the overall performance of GPU and CPU. We therefore evaluate the performance of int8 training on TensorFlow [24] using multi-threaded acceleration for CPU

and cuDNN [25] acceleration for GPU, as shown in brackets. It shows that DarkFPGA achieves 11 times speed up over CPU-based implementation and 3 times more energy efficient than GPU implementation on TensorFlow.

Finally, the area cost of the design targeting the MAX5 platform is shown in Table III. This demonstrates, also based on our design space exploration, that the performance of FPGA-based training deep neural network is limited by the number of DSPs available on the chip. If the parallelism level (T_B, T_I) increases to $(128, 128)$, FPGA may achieve similar performance as GPU, which may require over 16500 on-chip DSPs. This requirement is likely to be achieved by a single Intel Stratix 10 FPGA with setting $(T_B = 128, T_I = 128)$, or extending batch-level parallelism to 4 Xilinx ultrascale+ VU9P FPGAs each with $T_B = 32, T_I = 128$.

TABLE III
RESOURCE UTILIZATION OF THE INT8 DARKFPGA ACCELERATOR ON MAX5 PLATFORM

	LUT	DSP	BRAM18	URAM
Available	1182240	6840	4320	960
Utilization	678716	6241	1232	131
Percentage	57.41%	91.24%	28.52%	13.65%

VII. CONCLUSION

This work proposes DarkFPGA, a novel FPGA framework for efficient training of deep neural networks. The DarkFPGA accelerator explores batch-level parallelism, which provides efficient training acceleration for both forward and backward 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.

ACKNOWLEDGMENT

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1 and EP/P010040/1), Maxeler, Intel and Xilinx is gratefully acknowledged.

TABLE IV
PERFORMANCE COMPARISON OF THE FINAL FPGA DESIGN VERSUS CPU AND GPU.

	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 2018.2
Flag	-Ofast	-	-
Frequency	3.47GHz	1.58GHz	200MHz
Precision	32-bit floating point	32-bit floating point	8-bit fixed point
Technology	32nm	28nm	16nm
Processing Time per batch (ms)	66439 (3270)	126 (53.4)	288
Threads	1 (24)	-	-
Power (W)	131 (204)	187 (217)	12.4
Energy (J)	8712 (667.1)	23.6 (11.6)	3.6
Energy Efficiency	1x (13x)	369x (751x)	2440x

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, and M. Bernstein, "Imagenet large scale visual recognition challenge," *IJCV*, 2015.
[2] K. He, G. Gkioxari, P. Dollr, and R. Girshick, "Mask r-cnn," 2017.
[3] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, and S. Subhaschandra, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.
[4] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *arXiv:1802.07569*, 2018.
[5] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh, and H. Wu, "Mixed precision training," *arXiv:1710.03740*, 2017.
[6] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *arXiv:1805.11046*, 2018.
[7] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *arXiv:1803.03383*, 2018.
[8] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *arXiv:1802.04680*, 2018.
[9] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018.
[10] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *ASP-DAC*, 2016.
[11] J. Redmon, "Darknet: Open source neural networks in C," http://pjreddie.com/darknet/, 2013–2016.
[12] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, "Maximum performance computing with dataflow engines," in *High-performance computing using FPGAs*, 2013.
[13] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of CNN training on FPGA clusters," in *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.
[14] Y. Li and A. Pedram, "Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks," in *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017.
[15] S. Shao and W. Luk, "Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation," in *27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017.
[16] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-CNN: An fpga-based framework for training convolutional neural networks," in *IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016.
[17] S. J. W. Siddhartha and P. H. Leong, "Simultaneous inference and training using on-FPGA weight perturbation techniques," in *International Conference on Field-Programmable Technology (FPT)*, 2018.
[18] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the Intel HARPv2 Xeon+ FPGA platform: A deep learning case study," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.
[19] "Performance guide of using nchw image data format." [Online]. Available: https://www.tensorflow.org/guide/performance/overview#use_nchw_imag
[20] D. Steinkraus, I. Buck, and P. Simard, "Using GPUs for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 2005.
[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.
[23] S. Krishnan, P. Ratusziak, C. Johnson, D. Moss, and S. Subhaschandra, "Accelerator templates and runtime support for variable precision CNN," *CISC Workshop*, 2017.
[24] M. Abadi *et al.*, "TensorFlow: a system for large-scale machine learning," in *OSDI*, 2016.
[25] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv:1410.0759*, 2014.