F-CNN: An FPGA-based Framework for Training Convolutional Neural Networks

Wenlai Zhao*†‡, Haohuan Fu*†‡, Wayne Luk§, Teng Yu§, Shaojun Wang¶,
Bo Feng*, Yuchun Ma* and Guangwen Yang*†‡,
*Department of Computer Science and Technology, Tsinghua University, China
†Ministry of Education Key Laboratory for Earth System Modeling,
and Center for Earth System Science,Tsinghua University, China
‡Tsinghua National Laboratory for Information Science and Technology, China
§Department of Computing, Imperial College London, UK
¶Department of Automatic Test and Control, Harbin Institute of Technology, China
{zhao-wl11@mails.tsinghua.edu.cn, haohuan@mail.tsinghua.edu.cn}

***Abstract*—This paper presents a novel reconfigurable framework for training Convolutional Neural Networks (CNNs). The proposed framework is based on reconfiguring a streaming datapath at runtime to cover the training cycle for the various layers in a CNN. The streaming datapath can support various parameterized modules which can be customized to produce implementations with different trade-offs in performance and resource usage. The modules follow the same input and output data layout, simplifying configuration scheduling. For different layers, instances of the modules contain different computation kernels in parallel, which can be customized with different layer configurations and data precision. The associated models on performance, resource and bandwidth can be used in deriving parameters for the datapath to guide the analysis of design trade-offs to meet application requirements or platform constraints. They enable estimation of the implementation specifications given different layer configurations, to maximize performance under the constraints on bandwidth and hardware resources. Experimental results indicate that the proposed module design targeting Maxeler technology can achieve a performance of 62.06 GFLOPS for 32-bit floating-point arithmetic, outperforming existing accelerators. Further evaluation based on training LeNet-5 shows that the proposed framework achieves about 4 times faster than CPU implementation of Caffe and about 7.5 times more energy efficient than the GPU implementation of Caffe.**

I. Introduction

Convolutional Neural Network (CNN [1]) is one of the most successful deep learning models. FPGA-based designs [2]–[4] are proposed to accelerate CNN classification process (forward computation) and achieve considerable speedup and higher energy efficiency than CPU and GPU [5].

The training process of CNNs shares a similar computation pattern with classification, which shows the potential for accelerating the training process on FPGA-based platforms. However, the training process involves much more computation and more complicated workflow, which remains challenging for designing and accelerating the whole training process based on FPGAs. Specifically, the computation resources on modern FPGAs are too limited to implement the whole training process, which calls for improvements in task partition and scheduling. Moreover, classification is mainly based on specific pre-trained CNNs, while the training process should be more flexible to support different network configurations, which is challenging to FPGA-based hardware designs.

In this paper, we address the above problems and present an FPGA-based CNN training framework. In summary, the main contributions of this paper are as follows:

- A novel reconfigurable design for CNN training called F-CNN. It involves reconfiguring a streaming datapath at runtime to cover the training tasks for the various layers in a CNN. The streaming datapath contains various parameterized modules, which can be customized to produce implementations with different configurations.
- Analytical models for performance, bandwidth and resource usage of the modules. Given certain network configurations, these models can be used to estimate the implementation specifications and maximize performance under the constraints on bandwidth and hardware resources.
- Evaluation of the proposed F-CNN prototype targeting a Maxeler FPGA platform with Altera Stratix V FPGAs. The convolutional modules for AlexNet achieve 62.06 GFLOPS for 32-bit float, outperforming most existing accelerators. Overall evaluation based on training LeNet-5 shows about 4 times speedup over CPU-based implementations of Caffe and is about 7.5 times more energy efficient than GPU-based implementations of Caffe.

The organization of the paper is as follows. Section II reviews the CNN training process and some existing FPGA-based accelerators. Section III introduces the F-CNN design. Section IV presents the design of computation modules and the analytical models. Section V includes the experimental results. Section VI covers conclusions.

II. Background and Motivation

A. CNN Model

Figure 1 shows an typical LeNet-5 CNN [6] for handwriting digit recognition, which is simpler than many modern CNNs, but sufficiently representative for introducing the basic CNN models. It has 2 convolutional layers, 2 max-pooling layers and 2 fully connected multilayer perceptron (MLP [7]) layers.

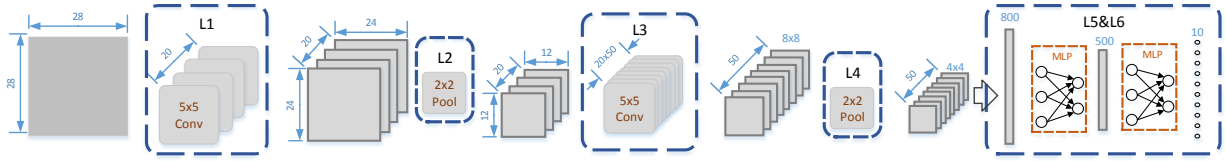

Fig. 1. A LeNet-5 CNN

TABLE I
CONFIGURATIONS OF LENET-5

	Ni	No	Ri	Ci	Ro	Co	K	S	P
L1	1	20	28	28	24	24	5	1	
L2	20	20	24	24	12	12			2
L3	20	50	12	12	8	8	5	1	
L4	50	50	8	8	4	4			2
L5	800	500							
L6	500	10							

1) Convolutional Layer: We define the following symbols to describe the layer, which are called **configurations** of a layer. The numbers of input and output images are Ni and No. The input image size is $Ri * Ci$ and the output image size is $Ro * Co$. There are $Ni * No$ convolutional filters, each of which connects one input image to one output image. The filter size is $K * K$ and the stride of convolution is S.

The convolutional operation can be described as (1) and the variables in the equation are defined as:

$$v_{i,j}^{m} = b^{m} + \sum_{n=0}^{Ni-1} \sum_{ii=0}^{K-1} \sum_{jj=0}^{K-1} w_{(K-1-ii),(K-1-jj)}^{m,n} \cdot u_{(i \cdot S+ii),(j \cdot S+jj)}^{n} \tag{1}$$

- $v_{i,j}^{m}$: the value at position (i, j) in the mth output image
- $u_{ii,jj}^{n}$: the value at position (ii, jj) in the nth input image
- $w_{ii,jj}^{m,n}$: the weight at position (ii, jj) in the convolution filter which connects the nth input image to the mth output image
- b^{m}: the *bias* of output image m

In the LeNet-5 example, L1 and L3 are convolutional layers and the configurations of them are shown in Table I.

2) Pooling Layer: Pooling layers, also known as the sub-sampling layers, are designed to shrink the feature images and reduce the redundancy in the features. We also define the numbers of input and output images as Ni and No with size $Ri * Ci$ and $Ro * Co$. The pooling size is defined as P. Max-pooling is the mostly used pooling layer in CNNs. In the example, L2 and L4 are max-pooling layers. The output of a max-pooling layer is calculated as (2), where v, u represents the values in output and input images. The configurations of L2 and L4 are shown in Table I.

$$v_{i,j}^{m} = \max_{0 \le ii,jj < P, n=m} u_{(i \cdot P+ii),(j \cdot P+jj)}^{n} \tag{2}$$

3) MLP Layer: In an MLP layer, the output images of the previous layer are expanded to a feature vector. We define the input vector as V_{in} and the output vector as V_{out}, with size Ni and No. The weight matrix is W, which has a size of $No * Ni$. The output of an MLP layer is calculated as (3), which is a typical matrix-vector multiplication and B is a bias vector with size No.

$$V_{out} = W \times V_{in} + B \tag{3}$$

In LeNet-5, L5 and L6 are MLP layers. The configurations are shown in Table I.

4) Activation Function: To involve nonlinearity into neural networks, activation functions are utilized to process the output data of each layer. Some typical activation functions are $tanh$, $sigmoid$ and $ReLU$. $Softmax$ is commonly used to calculate the posterior probability for logistic regression in the last layer.

The activation functions are monotone increasing, so if a convolutional layer is followed by a pooling layer, the activation function can be put after the pooling layer. In LeNet-5, there is no activation function in L1 and L3. $Tanh$ is used in L2, L4 and L5. $Softmax$ is used in L6.

B. Back-propagation Algorithm

The CNN model above is the forward computation. In the training process, back-propagation algorithm is a common method for weights adjustment in conjunction with an optimization method such as gradient descent. The basic idea of the back-propagation algorithm is to propagate the error (the *gradient* in gradient descent) back along the network and adjust the weights to correct the error.

In general, we assume there is a L-layer MLP. For each layer l ($1 \le l \le L$) in the network, a^{l-1} is the input feature vector, z^{l} is the output vector before activation function, w^{l} is the weight matrix, b^{l} is the bias vector and σ is the activation function. The back-propagation algorithm for training this neural network can be summarized into 3 steps.

1) Forward computation: For each layer l, calculate:

$$\begin{aligned} z^{l} &= w^{l} a^{l-1} + b^{l} \\ a^{l} &= \sigma(z^{l}) \end{aligned} \tag{4}$$

In layer L, a^{L} represents the classification result.

2) Error propagation: In supervised training, each input a^{0} has a target result t. We define the *loss function* C, which is a function of a^{L} and t. The error of the output layer L is:

$$\delta^{L} = \nabla_{a^{L}} C \odot \sigma'(z^{L}) \tag{5}$$

where $\odot$ is the element-wise multiplication, σ' is the derivative function of σ.

Then, for each layer l ($1 \leq l < L$), we can propagate the error as the following:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{6}$$

3) Weight update: Given the error δ^l for layer l, the partial derivatives of loss function C with respect to weights w^l and bias b^l are:

$$\begin{aligned} \Delta w^l &= \frac{\partial C}{\partial w^l} = a^{l-1}\delta^l \\ \Delta b^l &= \frac{\partial C}{\partial b^l} = \delta^l \end{aligned} \tag{7}$$

Then we can update the weights as:

$$\begin{aligned} \hat{w}^l &= w^l + \eta \Delta w^l \\ \hat{b}^l &= b^l + \eta \Delta b^l \end{aligned} \tag{8}$$

where η is called the *learning rate*, and $\hat{w}^l$, $\hat{b}^l$ are the updated weights and bias.

The back-propagation process above can be used for MLP layers in CNN. For a pooling layer, we can easily replicate one value in δ^{l+1} to a $P * P$ field in δ^l, and no weights update is involved. For a convolutional layer, we should change the matrix multiplication in equations (6) and (7) to the cross-correlation operation. The difference between cross-correlation and convolution is a time reversal on one the input data.

In brief, the backward process shares similar computation patterns with the forward process, but involves approximately two-fold of computation operations for both error propagation and weight update, which means we can use similar computation kernel design in both forward and backward modules with differences on data scheduling.

In practice, the most commonly used optimization method for CNN is called *stochastic gradient descent* (SGD) [8], which randomly choose a subset of the training data, called a *minibatch*, and update parameters based on the average error of the it. Therefore, the input data of a layer in CNN can be stored as a 4-dimension tensor, and the meaning of 4 dimensions are: number of feature images in 1 training sample (N), number of rows in one image (R), number of columns in one image (C) and the size of a minibatch (B_s).

C. Related Work

Most existing FPGA-based designs are focused on classification. Benkrid [9] proposed a 2D convolution core on FPGA in 2002. Later, Zhang [10] explored the different optimizations for FPGA-based convolution core on resource utilization, bandwidth, energy efficiency and memory access pattern. Recently, Chen [4] analyzed the design space and performance of a convolution layer based on the roofline model. Ovtcharov [11] demonstrated a 3x better performance on convolutional accelerator than [4], but did not provide more details on the design. Besides, implementations such as [12] are focused on designing single neuron on FPGA. While these works provide good ideas and results on optimizing the computation kernels, they are still far from a practical solution to perform CNN training.

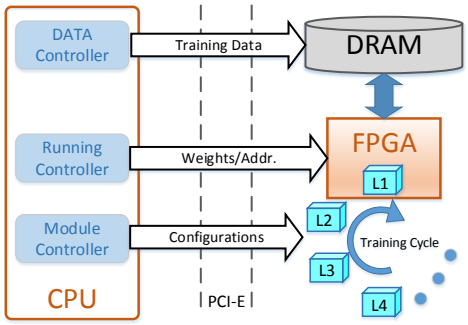

Fig. 2. The overall architecture of F-CNN

Some other works are more systematic. Work in [3] can implement a complete feedforward CNN on FPGA and can be used in some recognition applications such as face detection. Cadambi [2] designed an accelerator called MAPLE, which can handle matrix-multiplication based learning and classification computations, including convolution and MLP.

A few works implement back-propagation on FPGAs [13], [14]. However, they are designed for shallow neural networks and MLPs, but none of them is applicable to train CNN.

To our knowledge, our work is the first integrated CNN training framework on FPGA-based platform.

III. Framework Design

A. Design Principles

To design a framework for CNN training, we first consider the following principles.

1) Modularity: The hardware resources on a modern FPGA are far from enough to implement the whole training process. Therefore, a modular design is adopted in our framework. We partition the training process according to layers and provide parameterized modules, which can be customized to produce implementations with different configurations, for three kinds of computation layers (convolution, pooling and MLP).

2) Unified Datapath: In order to reduce the cost of module coupling and improve the efficiency of the whole framework, we design the modules to support a unified streaming datapath. As discussed, the data transferred between different layers are 4-dimension tensors. In our design, all tensors are stored in a consistent layout: $N * R * C * B_s$. We put B_s as the lowest dimension to increase data parallelism. To fit the unified streaming datapath, we design all the modules to support the same input and output data layout, so that we can access memory in efficient patterns (such as linear access pattern), which can maximize the utilization of the memory bandwidth.

3) Runtime Reconfiguration: Usually there are not enough FPGA accelerators to pre-configure all modules before execution. Therefore, we adopt runtime reconfiguration technology in the training workflow, which could make our framework applicable to platforms with one or more FPGA accelerators.

B. Architecture of F-CNN

The overall architecture of F-CNN is shown in Figure 2. The framework is designed based on a hybrid CPU/FPGA

platform, where CPU is the controller, FPGA is the computation accelerator and DRAM on the FPGA card is used for storing the input and output data of each computation module.

Module controller is responsible for customizing different computation modules based on the layer configurations and reconfiguring the modules into FPGA cards following a specific order, which is also the order of a *training cycle*: first the forward computation modules from the bottom to the top layer, and then the backward computation modules from the top to the bottom layer.

Data controller divides the training data into minibatches and loads them into the DRAM for training. For a multi-FPGA platform, the data controller also controls the data transfer between different FPGA cards.

Running controller calls the configured module to do the computation. The read/write addresses of data is passed as parameters, so that the module can access the data in DRAM. Besides the data, we also need to transfer the corresponding weights to the module. Compared with the size of training data, the size of weights is much smaller, and some of the weights need to be reversed in the back-propagation process according to the algorithm in Section II-B. Therefore, we store the weights and bias in CPU and transfer them to FPGA through PCI-E together with other parameters. This design can reduce the I/O load of DRAM, take fully advantage of the PCI-E bandwidth and make the module design independent of the weights. As a consequence, CPU is also responsible for the weight update (equation (8)) after receiving the partial derivatives from the FPGA.

In summary, given a real CNN, F-CNN first implements modules for each layer, then divides the training dataset into minibatches in CPU data controller. After that, the following iterative process is executed for training cycles:

1) Reconfigure a module into an FPGA card
2) Prepare data in DRAM, which contains three cases:
   2a. For the first module, training data is loaded from CPU to DRAM
   2b. For the following modules in a single-FPGA implementation, the intermediate data are already in DRAM and no transmission happens
   2c. For the following modules in a multi-FPGA implementation, intermediate data are transferred from the one FPGA card to another
3) Call the module to do the computation, with parameters and weights
4) Read back the results and update weights in CPU (for back-propagation modules) and goto step 1

IV. Module Design

The overall architecture of parameterized modules is shown in Figure 3. *Input controller* contains an address generator, which generates addresses to read the input data in each clock cycle, and a scheduler, which caches the input data and feed the data into computation kernels. *Output controller* has an address generator for storing output data into DRAM, and transfers the updated weights back to CPU through PIC-E.

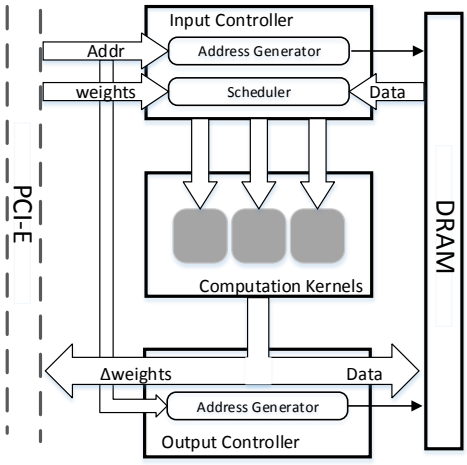

Fig. 3. The basic architecture of modules in F-CNN

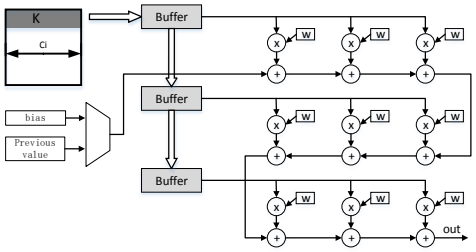

Fig. 4. Convolutional Kernel (K=3)

To guarantee the resource usage can fit into one FPGA card, we first design **kernels** for basic operations and implement one or more kernels in each module. The kernels can run in parallel, and the module performs the computation task by reusing the kernels with different data and weights, which is controlled by the scheduler in input controller. The resource usage of the modules with one kernel can be guaranteed to fit the hardware resources on one FPGA card. Increasing the number of kernels N_k can improve the resource utilization and the performance.

A. Forward Convolutional Kernel

Figure 4 shows the design of a convolutional kernel, which computes a $K * K$ convolution in each cycle.

A convolutional module requires 3 input streams (weights, bias and input images) and 1 output stream (output images). According to equation (1), there are three levels of accumulation for a final output image. The design shown in Figure 4 can implement the inner 2-level $K * K$ accumulation. In order to avoid wasting on-chip memory resources, we store the intermediate values of output images to DRAM and read back from DRAM to implement the outer accumulation. Therefore, there is one more input stream for intermediate values.

Here we use a linear memory access pattern to access data in DRAM. In order to get $K*K$ data in one image, a minimum buffer with size

$$BuffSize = Ci * K * B_s * DataLen \tag{9}$$

is required (for the gray area) in the scheduler, where $DataLen$ is the bit-width of data representation. By controlling the input data, we can guarantee the output data layout and store them to DRAM without buffering.

Based on the design, the following specifications are required to customize a convolutional module: layer configurations, size of minibatch B_s, data representation ($DataLen$), number of kernels N_k and running frequency F. In particular, B_s is usually a design parameter of a CNN. However, in FPGA implementations, to achieve efficient DRAM access, the B_s is chosen to meet the requirement of data alignment.

Given the specification definition, we build the analytical model to show the relationship between performance, bandwidth and hardware resource utilization.

1) Performance model: In each clock cycle, a kernel can finish a $K*K$ convolution operation (here we do not consider the depth of the design pipeline). Each value in the output image requires Ni convolution operations. Therefore, for a module with N_k kernels, the total clock cycles and the running time T under frequency F are:

$$\begin{aligned} Cycles &= Ni \times No \times Ro \times Co \times B_s/N_k \\ T &= Cycles/F \end{aligned}$$

One convolution operation involves $K*K$ multiplication addition (MA) operations. The total number of operations is:

$$N_{op} = 2 \times Ni \times No \times Ro \times Co \times K \times K \times B_s$$

If we use 32-bit float as the data precision, the theoretical performance (FLOPS) can be calculated as:

$$Perf = N_{op}/T = 2 \times N_k \times F \times K^2 \tag{10}$$

2) Bandwidth model: There are two streams from PCI-E and three streams from DRAM connected to the module. In each cycle of the fully-pipelined design, one value is read from the weights and bias stream. Meanwhile, $S*N_k$ values are read from the input data stream, N_k values are read from the intermediate data stream, and N_k values are stored to DRAM via the output image stream. Therefore, the theoretical maximum bandwidth requirements of PCI-E and DRAM are:

$$\begin{cases} BW_{pcie} &= 2 \times DataLen \times F \\ BW_{dram} &= (2+S) \times N_k \times DataLen \times F \end{cases} \tag{11}$$

3) Resource model: There are four kinds of hardware resources on FPGA: LUT, FF, Block-RAM (BRAM) and DSP. We give an utilization estimation to each of them.

The resource consumption of address generator and stream control units is independent of N_k. We can define them as LUT_{io}, FF_{io}, $BRAM_{io}$ and DSP_{io}.

The major resource of the scheduler is an input buffer, which consumes BRAM, defined as $BRAM_{buff}$. According to equation (9), if we define the size of a BRAM block as S_{block}, then $BRAM_{buff} = \lceil Ci*K*B_s*DataLen/S_{block} \rceil$.

For a certain data precision and a certain FPGA platform, the resource consumption of an MA operation can be considered as constant, defined as LUT_{ma}, FF_{ma}, $BRAM_{ma}$ and DSP_{ma}. In the convolutional kernel, K^2 MAs are implemented and K^2 buffers are designed for caching the weights.

Therefore, the total resource usage of a convolutional module can be estimated as:

$$\begin{cases} LUT &= N_k \times K^2 \times LUT_{ma} + LUT_{io} \\ FF &= N_k \times K^2 \times FF_{ma} + FF_{io} \\ BRAM &= N_k \times K^2 \times (BRAM_{ma}+1) \\ & \quad + BRAM_{io} + BRAM_{buff} \\ DSP &= N_k \times K^2 \times DSP_{ma} + DSP_{io} \end{cases} \tag{12}$$

We put the equations (10), (11), (12) together as an analytical model for a forward convolutional module. Given the layer configurations, we can find the optimal N_k to achieve best performance, while not exceeding the limitation of bandwidth and available hardware resources.

B. Backward Convolutional Kernel

A backward convolutional kernel has two tasks: to calculate error δ and to calculate partial derivatives Δw and Δb, both of which are cross-correlation operations. We can implement the cross-correlation operation by reverse the data in the convolution kernel. Therefore, the backward computation kernel can also be designed as figure 4, but the scheduler is different from the forward module.

The difference on resource consumption is that we need two input buffers for both δ^l and δ^{l-1} when calculating the derivatives. The direction of I/O streams changes in different computation tasks. For example, when calculating the error, the weights need to be transferred from CPU to FPGA, but when calculate the derivatives, we need to transfer the Δw and Δb from FPGA to CPU. However, that will not affect the bandwidth.

In summary, we can build the analytical model for a backward convolutional module as follows:

$$\begin{cases} Perf &= 2 \times N_k \times F \times K^2 \\ BW_{pcie} &= 2 \times DataLen \times F \\ BW_{dram} &= (2+S) \times N_k \times DataLen \times F \\ LUT &= N_k \times K^2 \times LUT_{ma} + LUT_{io} \\ FF &= N_k \times K^2 \times FF_{ma} + FF_{io} \\ BRAM &= N_k \times K^2 \times (BRAM_{ma}+1) \\ & \quad + 2 \times BRAM_{buff} + BRAM_{io} \\ DSP_{conv} &= \times N_k \times K^2 \times DSP_{ma} + DSP_{io} \end{cases} \tag{13}$$

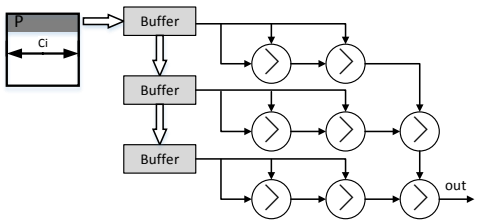

Fig. 5. Max-pooling Kernel (P=3)

C. Pooling Kernels

The design of a forward max-pooling kernel with pooling size P is shown in Figure 5. To support the data layout, a buffer for the input data with the size $P * Ci * B_s * DataLen$ is required in the scheduler. Besides, $P^2 - 1$ comparators are implemented in a pooling kernel. Two DRAM streams are required for the input and output data. The backward pooling kernel replicates an input value to a $P * P$ field. A $Co * B_s * DataLen$ buffer and two DRAM streams are required.

A forward pooling layer usually followed by an activation function. To implement an activation function in the pooling kernel, one more DRAM stream for the function output (a^l in equation (4)) is required. Similarly, we implement the derivative function ($\sigma^{'}(z^l)$) in the backward pooling module and a stream for z^l (equation (6)) is required. Therefore, there are three DRAM streams in pooling modules.

Usually, the resource consumption of comparison and activation functions is not high (see Table III), and it is obvious that bandwidth is the bottleneck of pooling modules.

$$BW_{dram} = (2 + P) \times N_k \times DataLen \times F \tag{14}$$

D. MLP Kernels

As discussed in II-B, the major computation of both forward and backward MLP layer is vector-matrix multiplication. Because we use a minibatch for training, the real computation executed in MLP layer is a matrix-matrix multiplication (MM).

To deal with MMs with varied size, we design the MLP module based on a blocked matrix multiplication (BMM) algorithm [15] (Algorithm 1).

Algorithm 1 Blocked Matrix Multiplication (BMM)

1: Transform input matrix A ($M \times N$), weight matrix B ($N \times O$) and output matrix C ($M \times O$) into $R * R$ blocks
2: Let $A(i, j)$, $B(i, j)$ and $C(i, j)$ be the ith row block and jth column block in A, B and C
3: **for** i from 1 to M/R **do**
4: **for** j from 1 to O/R **do**
5: Initialize $B(i, j)$ with 0
6: **for** k from 1 to N/R **do**
7: $C(i, j)_k = A(i, k) \times B(k, j)$
8: $C(i, j) = C(i, j)_k + C(i, j)$
9: **end for**
10: Output $C(i, j)$
11: **end for**
12: **end for**

In our module, the computation kernel is an $R * R$ MM unit. It is worth noting that to support the unified data layout, which is considered as a row-major layout for the matrix in DRAM, we adopt 2D-stride memory access pattern in the address generator to access the blocks. Besides, we add activation functions in the module to accomplish a complete MLP layer.

Two input buffers with size $N * R * DataLen$ is required in the scheduler to cache one row of blocks in matrix A and one column of blocks in matrix B. One output buffer (**line 8**) with size $R^2 * DataLen$ is required in each kernel. In forward MLP module, there are two PCI-E streams for weights and bias and three DRAM streams for input data, output data and results of activation functions. The streams in backward module have the same number but different direction.

In summary, the analytical model for MLP modules is:

$$\begin{cases} Perf &= 2 \times N_k \times F \times R^2 \\ BW_{pcie} &= 2 \times DataLen \times F \\ BW_{dram} &= 3 \times N_k \times DataLen \times F \\ LUT &= N_k \times LUT_{kernel} + LUT_{io} \\ FF &= N_k \times FF_{kernel} + FF_{io} \\ BRAM &= N_k \times BRAM_{kernel} \\ & \quad +2 \times BRAM_{buff} + BRAM_{io} \\ DSP &= N_k \times DSP_{kernel} + DSP_{io} \end{cases} \tag{15}$$

where

$$\begin{cases} LUT_{kernel} &= R^2 \times LUT_{ma} + LUT_{act} \\ FF_{kernel} &= R^2 \times FF_{ma} + FF_{act} \\ BRAM_{kernel} &= R^2 \times BRAM_{ma} + BRAM_{act} \\ & \quad + \lceil \frac{R^2 \times DataLen}{S_{block}} \rceil \\ DSP_{kernel} &= R^2 \times DSP_{ma} + DSP_{act} \\ BRAM_{buff} &= \lceil \frac{N \times R \times DataLen}{S_{block}} \rceil \end{cases} \tag{16}$$

E. Model modification

1) Data Reuse: In bandwidth analysis, we consider the bandwidth required by all kernels, which is an upper bound and can be significantly reduced in practice by designing data reuse in the scheduler. For example, with the buffer size in equation (9), one input data is used in K^2/S^2 MA operations during its lifetime in the buffer. However, we can see from equation (1) that one input data can be used in at most $No * K^2/S^2$ MA operations. Therefore, if a specific implementation is bandwidth bounded, we can extend the lifetime of the data by designing larger buffers (consuming more BRAM), so that to improve the data reuse and reduce the bandwidth.

2) Runtime Reconfiguration: In performance analysis, we only consider the execution cycles of a module. However, as we use runtime reconfiguration in the framework, the actual execution time is: $T_{exec} = T + T_{reconfig} + T_{datatrans}$. If using one FPGA accelerator, we do not need to transfer data between FPGAs, so $T_{exec} = T + T_{reconfig}$. If there are multiple FPGA accelerators, we can do the reconfiguration on one FPGA while others are working. On the other hand, we need to consider the data transfer, then $T_{exec} = T + T_{datatrans}$. The implementation strategy is chosen based on hardware platform ($T_{reconfig}$) and problem size (T and $T_{datatrans}$), which will be discussed in Section V.

V. Experiment results

A. Platform-specific implementation

We develop a prototype for F-CNN based on Maxeler technology [16]. The MaxCompiler tool chains allow us to do hardware programming with a high-level language and the CPU controller is programmed in C++.

TABLE II
ANALYSIS AND IMPLEMENTATION RESULTS OF ALEXNET (32-BIT FLOAT, B_s=96, F=150MHZ, TIME IN ms, PERFORMANCE IN $GFLOPS$)

Layers	Analysis			FPGA Implementation			CPU Implementation (Caffe)		FPGA2015 [4]
	$\bar{N}_k$	Performance	Resource bound	N_k	Time	Performance	Time	Performance	Performance
L1	2.4	81.76	BRAM	3	113.6	89.04	1082.1	9.34	27.50
L2	13.9	79.08	BRAM	12	321.9	66.79	2267.3	9.48	83.79
L3	21.3	43.36	BW_{DRAM}	36	262.8	54.62	1801.1	7.97	78.81
L4	21.3	43.36	BW_{DRAM}	36	198.6	54.20	1415.1	7.61	77.94
L5	21.3	43.36	BW_{DRAM}	36	133.0	53.98	986.5	7.27	77.61
Overall					1029.9	62.06	7552.1	8.46	61.62

TABLE III
RESOURCE CONSUMPTION AND LIMITATION ON STRATIX V

	LUT	FF	DSP	BRAM
Stream ctrl units	57700	77600	0	460
32-float ADD	600	650	0	3
32-float MUL	150	400	1	2
32-float CMP	60	30	0	0
$tanh$ (32-float)	5600	6400	10	22
$tanh'$(32-float)	5400	2500	12	23
Available	524800	1049600	1963	2567
PCI-E maximum bandwidth	8 GB/s			
DRAM maximum bandwidth	38.4 GB/s			

TABLE IV
CONFIGURATION OF CONVOLUTIONAL LAYERS IN ALEXNET

	Ni	No	Ri	Ci	Ro	Co	K	S
L1	3	48	227	227	55	55	11	4
L2	48	128	31	31	27	27	5	1
L3	256	192	15	15	13	13	3	1
L4	192	192	15	15	13	13	3	1
L5	192	128	15	15	13	13	3	1

The experimental platform is a Maxeler MPC-X dataflow node which has 8 Maia accelerator cards. Each Maia card has an Altera Stratix V (5SGSD8) FPGA and 48GB DDR3 DRAM. The maximum bandwidth of DRAM is 38.4 GB/s. The FPGA cards are connected to the CPU via PCI Express 2.0 and the maximum PCI-E bandwidth is 8 GB/s.

All software results in our experiment are obtained by Caffe [17] running on an Intel Xeon E5-2680 v3 CPU (12 cores, 2.50GHz) with an NVIDIA Tesla K20X GPU accelerator (2688 CUDA cores). The Stratix V FPGA and K20X GPU use the same 28nm transistor technology and Xeon E5-2680 CPU uses 22nm transistor technology.

B. Module Customization

To validate the analytical model, we first test some basic resource consumption on Stratix V, shown in Table III. Based on the model, we calculate the optimal $\bar{N}_k$ for different convolutional layers in AlexNet [1] (Table IV shows the configurations). We use 32-bit float data precision, B_s is 96 and the frequency is 150 MHz.

In practice, some hardware implementation issues should be considered, such as timing issues, data alignment and compiler optimization. Therefore, the actual N_k is chosen to be able to implement on hardware and as close as possible to $\bar{N}_k$. Table II shows the analytical results and the implementation results of the AlexNet layers.

In L1, N_k is bigger than $\bar{N}_k$ because of the resource optimization in the compiler. N_k in L2 – L5 are chosen to meet the requirement of data alignment for the input/output streams. On Maxeler platform, the data-width of a stream should be either a divisor or a multiple of 1536 bits. Otherwise, extra resources will be consumed in the stream control units and will affect the efficiency of the hardware design. According to the analysis results, L3, L4 and L5 are bounded by the DRAM bandwidth. So we adopt data reuse by doubling the buffer size. The final resource bound is BRAM.

Compared with a multi-thread CPU implementation, our convolutional module can achieve about 7 times speedup on average. Compared with a recently proposed FPGA-based convolution design on Xilinx Virtex 7 [4], our module can achieve slightly better performance with no need for extra memory control units.

C. Framework Evaluation

We implement and train the LeNet-5 CNN with the configurations in Table I. We use 32-bit float data precision, B_s is 384 and running frequency is 150 MHz. There are 6 layers in LeNet-5, so 12 modules are implemented in the training datapath. To achieve better performance for small size problems ($No = 10$ in L6), we set the MLP kernel size R to 4. The dataset is MNIST for handwriting digits recognition, which has a training set of 60000 examples and a test set of 10000 examples. We divide the training dataset into 150 minibatches. The training process runs for 200 iterations.

The execution time (running and data transfer time) per iteration of each module is shown in Table V. The reconfiguration time ($T_{reconfig}$) on Maia accelerator is not uniform (varying from 100ms to 1sec) and single-FPGA implementation is inefficient as discussed in Section IV-E2. We use two FPGA cards in our implementation to overlap the reconfiguration time on one card and the execution time on the other.

As shown in Table V, F-CNN achieves around 4 times speedup over a CPU-based implementation and is comparable to a GPU-based implementation on overall performance. Since we use a heterogeneous platform, we compare the power consumption of FPGA and GPU accelerators. The average power consumption (27.3Watt) of FPGA is obtained by the Maxeler performance monitoring tools. That is approximately

TABLE V
AVERAGE EXECUTION TIME (SEC) PER ITERATION OF LENET-5 ON F-CNN, CPU, AND GPU

Layers	N_k		F-CNN		Caffe (CPU)		Caffe (GPU)	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
L1(Conv)	12	12	0.59	1.21	6.84	7.70	2.55	7.19
L2(Pool)	24	24	0.53	0.57	1.85	0	0.04	0
L3(Conv)	12	12	4.67	10.32	36.37	33.11	5.33	3.38
L4 (Pool)	24	24	0.17	0.18	1.05	0	0.01	0
L5 (MLP)	24	24	0.92	1.82	2.04	2.02	0.05	0.4
L6 (MLP)	24	24	0.18	0.20	0.08	0.07	0.03	0.02
Total (Speedup)			21.4(4.3×)		91.1 (1×)		18.7 (4.9×)	
Power (Energy Efficiency)			27.3Watt (7.5×)				235Watt (1×)	

equal to the power consumption of one FPGA because at any given time, only one of the FPGAs is running and the other is reconfiguring or waiting. The power of K20X GPU is 235 Watt. By multiplying time and power consumption, we can see that our FPGA-based design is 7.5 times more energy efficient than the GPU implementation of Caffe.

According to Table III, floating-point MA operation consumes more BRAM on Stratix V, which restricts the performance in our current implementation. Improvements can be made by using fixed point data precision and by targeting our design to more advanced FPGAs like Arria 10 (which has hardened floating-point DSPs) and Stratix 10 (which has more available hardware resources). Moreover, we can make higher resource utilization with better runtime reconfiguration technologies to achieve higher performance.

VI. CONCLUSION

We propose F-CNN, a novel reconfigurable framework for training convolutional neural networks. F-CNN involves reconfiguring a streaming datapath at runtime to cover the training cycle for the various layers in a CNN. The datapath of F-CNN contains various modules, which are designed to be customizable. Analytical models are developed for customizing modules to maximize performance under the constraints of bandwidth and hardware resources. We implement AlexNet and LeNet-5 to evaluate the modules and framework. Experiment results show higher performance than CPU and higher energy efficiency than GPU, which illustrates the advantages of employing FPGA technology as heterogeneous accelerators on HPC platforms.

As a prototype, F-CNN shows the feasibility of training CNNs on FPGA-based platforms deploying runtime reconfiguration. There are many opportunities for further work, for example, to explore further optimizations of modules with different data precisions, to achieve more effective software-hardware co-design.

ACKNOWLEDGEMENT

This work is supported in part by National Natural Science Foundation of China (Grant No. 61303003, 41374113), by Tsinghua University Initiative Scientific Research Program (Grant No. 20131089356), by European Union Horizon 2020 Programme with project reference 671653, by UK EPSRC (grant reference EP/I012036/1 and EP/L00058X/1), and by the Maxeler University Program and by Altera.

REFERENCES

[1] Krizhevsky, A., Sutskever, I., Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[2] Cadambi, S., Majumdar, A., Becchi, M., Chakradhar, S., Graf, H. P. (2010, September). A programmable parallel accelerator for learning and classification. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (pp. 273-284). ACM.

[3] Chakradhar, S., Sankaradas, M., etc. (2010, June). A dynamically configurable coprocessor for convolutional neural networks. In ACM SIGARCH Computer Architecture News (Vol. 38, No. 3, pp. 247-257). ACM.

[4] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J. (2015, February). Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 161-170). ACM.

[5] Yadan, O., Adams, K., Taigman, Y., Ranzato, M. A. (2013). Multi-gpu training of convnets. arXiv preprint arXiv:1312.5853, 17.

[6] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[7] Crone, S. F. (2005). Stepwise selection of artificial neural network models for time series prediction. Journal of Intelligent Systems, 14(2-3), 99-122.

[8] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010 (pp. 177-186). Physica-Verlag HD.

[9] Benkrid, K., Belkacemi, S. (2002, November). Design and implementation of a 2D convolution core for video applications on FPGAs. In Digital and Computational Video, 2002. DCV 2002. Proceedings. Third International Workshop on (pp. 85-92). IEEE.

[10] Zhang, H., Xia, M., Hu, G. (2007). A multiwindow partial buffering scheme for FPGA-based 2-D convolvers. Circuits and Systems II: Express Briefs, IEEE Transactions on, 54(2), 200-204.

[11] Ovtcharov, K., Ruwase, O., Kim, J. Y., Fowers, J., Strauss, K., Chung, E. S. (2015). Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper, 2.

[12] Coric S. (2000). A neural network FPGA implementation. In Neural Network Applications in Electrical Engineering, 2000. NEUREL 2000. Proceedings of the 5th Seminar on (pp. 117-120). IEEE.

[13] Omondi, A. R., Rajapakse, J. C. (Eds.). (2006). FPGA implementations of neural networks (Vol. 365). New York, NY, USA:: Springer.

[14] Girones, R. G., Palero, R. C., (2005). FPGA implementation of a pipelined on-line backpropagation. Journal of VLSI signal processing systems for signal, image and video technology, 40(2), 189-213.

[15] Matam K K, Prasanna V K. Energy-efficient large-scale matrix multiplication on FPGAs[C]//Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on. IEEE, 2013: 1-8.

[16] O. Pell and V. Averbukh, Maximum Performance Computing with Dataflow Engines. Computing in Science & Engineering, pp. 98-103, 2012.

[17] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T. (2014, November). Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the ACM International Conference on Multimedia (pp. 675-678). ACM.