2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates

Yijin Guan[1,3*], Hao Liang[2,3*], Ningyi Xu[3], Wenqiang Wang[3], Shaoshuai Shi[3],
Xi Chen[3], Guangyu Sun[1,5], Wei Zhang[2] and Jason Cong[4,5,1†]
[1]Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China
[2]Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, China
[3]Microsoft Research Asia, Beijing, China
[4]Computer Science Department, University of California, Los Angeles, USA
[5]PKU/UCLA Joint Research Institute in Science and Engineering

***Abstract*—DNNs (Deep Neural Networks) have demonstrated great success in numerous applications such as image classification, speech recognition, video analysis, etc. However, DNNs are much more computation-intensive and memory-intensive than previous shallow models. Thus, it is challenging to deploy DNNs in both large-scale data centers and real-time embedded systems. Considering performance, flexibility, and energy efficiency, FPGA-based accelerator for DNNs is a promising solution. Unfortunately, conventional accelerator design flows make it difficult for FPGA developers to keep up with the fast pace of innovations in DNNs.**

To overcome this problem, we propose FP-DNN (Field Programmable DNN), an end-to-end framework that takes TensorFlow-described DNNs as input, and automatically generates the hardware implementations on FPGA boards with RTL-HLS hybrid templates. FP-DNN performs model inference of DNNs with our high-performance computation engine and carefully-designed communication optimization strategies. We implement CNNs, LSTM-RNNs, and Residual Nets with FP-DNN, and experimental results show the great performance and flexibility provided by our proposed FP-DNN framework.

I. Introduction

DNNs have brought in profound and revolutionary changes to the realm of artificial intelligence, and achieved great improvements in many domains such as computer vision [22] [13] [15], speech recognition [9], natural language processing [20], etc. Inspired by the impressive breakthroughs achieved by DNNs, many researchers in both academia and industry are longing to solve their problems with powerful DNNs. With their model accuracy closer to or even better than human, DNNs are widely deployed at scale in data centers, as well as in embedded systems like mobile phones and robots.

DNNs are well-known to be computation-intensive and memory-intensive because of their deep topological structures, complicated neural connections, and massive data to process. Due to these characteristics, it is challenging to achieve high performance and good energy efficiency when mapping DNNs onto generic computing system. To solve this problem, many hardware accelerators for DNN inference have been investigated recently. Among these designs, FPGA-based accelerators have gained great popularity because of their outstanding flexibility, performance and energy efficiency.

Unfortunately, hand-coded FPGA-based accelerators face both productivity and programmability challenges for mapping DNNs in real applications. On the one hand, the design and optimization of FPGA-based accelerators require much experience and expertise. It may cost a professional hardware developer several weeks to map a DNN model onto FPGAs, even with the help of high-level synthesis tools. For DNN designers, there are no programming interfaces or libraries (like cuBLAS and cuDNN in NVDIA GPUs) to easily map their model onto FPGAs. On the other hand, prior work on FPGA-based accelerators for DNNs focused on accelerating certain type of layers [28] or certain models [23] [19]. Since DNNs evolves rapidly, various model structures and optimization techniques are emerging so fast that re-designing FPGA-based accelerator for every new model or technique is quite inefficient.

According to the analysis above, there is a strong demand for an easy-to-use framework that can automatically map DNNs onto FPGAs. In this paper, we propose FP-DNN, which takes symbolic descriptions (in TensorFlow) of DNNs as input, and outputs implementations of the corresponding FPGA-based accelerators for model inference. We implement accelerators with RTL-HLS hybrid templates, and convert model inference into general-purpose computations like matrix multiplication. Several optimization kernels are developed and invoked to ensure the functionality, performance and energy efficiency of the accelerator. The entire compilation procedure is end-to-end and automated, which makes it possible for all DNNs researchers and users to use FPGA as a powerful device to perform model inference.

We make the following contributions in this paper:

*Yijin Guan and Hao Liang contributed equally to this work.

†In addition to being a faculty member at UCLA, Jason Cong is also a co-director of the PKU/UCLA Joint Research Institute and a visiting chair professor of Peking University.

This research was performed while Yijin Guan, Hao Liang, Shaoshuai Shi and Xi Chen were interns at Microsoft Research Asia.

978-1-5386-4037-1/17 $31.00 © 2017 IEEE
DOI 10.1109/FCCM.2017.25

- We build a framework that automatically maps DNNs onto FPGAs for model inference. Compared with previous accelerating work, this automated framework can save design time significantly.
- We divide the operations involved in model inference into computation-intensive part and layer-specific part. We implement high-performance matrix multiplication kernel for the computation-intensive part, and carefully optimize communication bandwidth for the layer-specific part. FP-DNN automatically generates the hardware implementation with RTL-HLS hybrid templates.
- Our framework can support almost all types of DNNs, and we implement several DNNs (CNNs, LSTM-RNNs, and Residual Nets) as case studies. FPGA-based accelerators generated by this framework can achieve good performance and energy efficiency. To the best of our knowledge, this is the first literature to implement ResNet-152 on FPGA. Such a design has demonstrated flexibility, scalability and productivity of our FP-DNN framework.

The rest of this paper is organized as follows: Section II reviews some related work on DNNs and FPGA-based automated frameworks. Section III describes the architecture of our proposed FP-DNN framework. Then, the hardware implementation details are provided in Section IV. In Section V, we show the experimental setup and results of our case studies. At last, Section VI concludes this paper.

II. Related Work

A. Deep Neural Networks

DNNs have evolved into a big community, and many interesting and powerful models have been proposed. They have achieved great success in computer vision, speech recognition, scene analysis, etc. Typically, DNNs can be divided into several categories. By topological structure, we can divide these models into Artificial Neural Networks, Recurrent Neural Networks, Residual Nets, etc. All these models are comprised of several neural layers, so by type of layers, there are convolutional layers, LSTM layers, fully-connected layers, recurrent layers, pooling layers, activation layers, etc. A single DNN can choose any topological structure mentioned above, and it may includes several types of layers in its configuration. So this results in a huge design space of possible model structures.

Currently, many open-source frameworks have been released for DNN research: TensorFlow [5], Caffe [14], Theano [6], Torch [4], CNTK [3], etc. TensorFlow is one of the most popular DNN frameworks. It constructs DNNs in python/c++ front-end as a data flow graph with a operation library, and performs computation on the graph. TensorFlow support various types of DNNs (ANN/CNN/RNN/...) as well as other scientific computation. We appreciate the concepts of tensor and data flow graph in TensorFlow, and choose TensorFlow as the high-level descriptions for DNNs in FP-DNN.

B. FPGA-based Automated Frameworks

Accelerating the inference phase of DNNs on FPGAs has been a hot research topic, and many automation tools or frameworks have also been proposed. Among these designs, [16] [21] [27] and [25] are four representatives.

In [16], Mahajan et.al proposed TABLA, a template-based framework for accelerating statistical machine learning. They focus on accelerating the training phase by automatically generating the corresponding accelerators for stochastic gradient descent with Verilog-based templates. [21] proposed DNNWEAVER, a framework that automatically generates a synthesizable accelerator for a given (DNN, FPGA) pair. And they generates accelerators using hand-optimized design templates (RTL-based). In [27], Zhang et.al proposed Caffeine, a hardware/software co-designed library to accelerate convolutional neural networks on FPGAs. And they propose to accelerate convolutional layers and fully-connected layers with a uniformed representation. [25] proposed DeepBurning, an automation tool to generate FPGA-based accelerators for NN models. DeepBurning compiles DNNs described in a Caffe-like script and generates the corresponding RTL-level accelerator under user-specified constraints.

III. Framework

A. Overview

The overall FP-DNN framework is shown in Figure 1. Model description, usually in the format of protobuf generated by TensorFlow, is fed into our *Symbolic Compiler*. The compiler generates C++ program and FPGA programming bitstream, which are executed by the *Host* and *Device* respectively for model inference. Inside the *Symbolic Compiler*, *Model Mapper* analyzes the model description, and extracts topological structure and operations of the target model. After optimizations and parameterization for the hardware implementation, *Model Mapper* outputs the hardware kernel schedule and kernel configuration to the code generators. *Software Generator* uses kernel schedule to generate the host code in C++. The host code is compiled by commercial C++ compiler to generate host programs. With kernel configuration, *Hardware Generator* generates the device codes by instantiating RTL-HLS hybrid templates. Commercial synthesis tools compile these hardware codes to get the programming file for final hardware implementation. The whole FP-DNN framework works in an "end-to-end" manner: from software-based model descriptions to FPGA-based model inference implementations. This procedure is all done automatically without any human intervention.

B. Model Mapper

Model Mapper analyzes model description to map the model onto hardware platform, and it generates the schedule and configuration for hardware kernels. Figure 2 shows an example of the working flow of *Model Mapper*. Figure 2a shows the example python code snippet in TensorFlow describing a CNN model, which contains three convolution layers. The pooling and activation layers are omitted for simplicity. The corresponding *Data Flow Graph* generated and executed by TensorFlow is shown in Figure 2b, where computation and data are shown as operating nodes and tensors respectively.

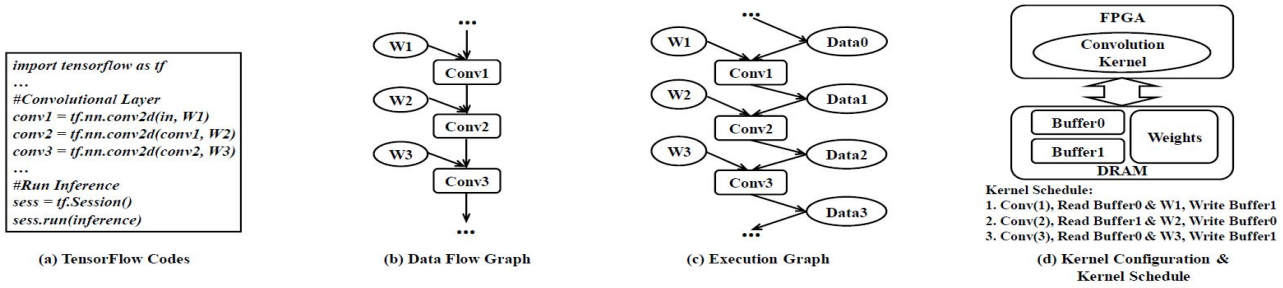


Fig. 2: Working Flow of Model Mapper

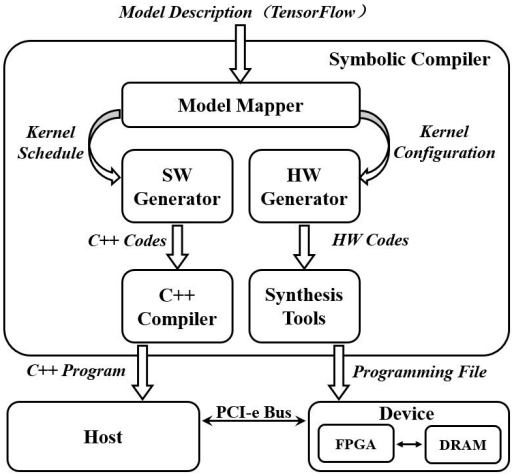


Fig. 1: FP-DNN Framework

The *Model Mapper* uses the model description to extract information about model structure and configurations of each layer. Although storing model parameters and intermediate results in on-chip BRAM can significantly improve performance, we do not have enough on-chip BRAMs on a single FPGA to store all of them for modern DNNs. As a result, we have to allocate data buffers in off-chip DDR memory for storing intermediate activations and model parameters. Then, *Model Mapper* generates an *Execution Graph* shown in Figure 2c, which shows ideally how the model inference is performed on hardware.

However, this *Execution Graph* can not be mapped onto FPGAs directly due to the limitation on computation resource and DRAM storage of modern FPGAs. So we propose to adapt resource reuse strategies to allocate hardware resources reasonably. To reuse computation resource, *Model Mapper* allocates only one hardware kernel, which will perform model inference layer by layer. Thus, in the example shown in Figure 2, only one convolution kernel is allocated. For storage resource reuse, *Model Mapper* allocates several physical buffers in DRAM as a memory pool, and we aim to minimize the number of physical buffers in final implementation. We formulate the data buffer reuse problem as a graph coloring problem.

During model inference, each data buffer has a range of time during which its contents must be kept intact. Thus, any two data buffers whose life spans intersect can not be placed in the same physical buffer. We construct an interval graph in which each vertex represents a data buffer. For any two data buffers whose life spans intersect, we connect their vertexes with an edge. We need to color this graph with minimum number of distinct colors so that no adjacent nodes are assigned the same color, which indicates that the data buffers with the same color can be assigned to the same physical buffer. The coloring problem for an interval graph can be solved optimally by left-edge algorithm in polynomial time. An algorithm description for applying left-edge algorithm in *Model Mapper* for physical buffer allocating is shown in Algorithm 1.

Algorithm 1: Physical Buffer Allocating Algorithm

Input: Initial Data Buffer Graph(G), and Data Buffers(V)
Output: Physical Buffer Allocations
Denote the left-edge and right-edge of the interval corresponding to data buffer v_i's life span as l_i and r_i respectively;
Sort V in ascending order of left-edge to get V';
# of physical buffers = 1;
while *not all* v *in* V' *have been allocated* **do**
  R = 0;
  while $\exists$ v_i *in* V' *with* $l_i > R$ **do**
    v_x = first v in V' with $l_x > R$;
    $R = r_x$;
    allocate v_x with current physical buffer;
    $V' = V' - v_x$;
  # of physical buffers += 1;

With carefully designed resource allocation strategies, *Model Mapper* outputs the kernel schedule and kernel configuration, which are shown in Figure 2d. In this example, only one convolution kernel will be allocated for computation, and two physical buffers are allocated for intermediate data storing.

C. SW Generator and HW Generator

SW Generator takes kernel schedule to generate the C++ codes for *Host*, which are in charge of kernel execution scheduling, model initializing, data buffer managing, etc. *SW Generator* instantiates a host code template with some key parameters extracted by *Model Mapper*, like the number of kernels, the number of physical buffers, kernel execution order, and so on. This host code is written in C++, and can be compiled by any commercial C++ compiler.

HW Generator is in fact a library of RTL-HLS hybrid templates for various types of layers. We use RTL-HLS hybrid templates instead of pure RTL templates or pure HLS templates for the following reasons: Compared with HLS, RTL designs usually utilize resources more efficiently, but it is well-known that RTL design is quite hard and time-consuming. HLS tools receive designs programmed in high-level programming languages (C, C++, OpenCL, etc.), then compile them into FPGA programming files. HLS design has a better abstraction for external modules or interfaces (like off-chip DRAM), which makes it easier and faster to implement complex control logics. However, currently HLS designs cannot explore as much fine-grained optimization as those in RTL designs.

To fully utilize the advantages of both design approaches, we take an RTL-HLS hybrid approach for template design: we use RTL for designing a high-performance computation engine, and we use the OpenCL-based HLS framework to implement the control logics for the RTL part. With the kernel configuration generated by *Model Mapper*, *HW Generator* instantiates the corresponding optimized kernel template to generate the hardware codes for *Device*. The RTL part of these kernel templates are written in Verilog, and the HLS part is written in OpenCL-based HLS. The generated hardware codes can be compiled by commercial synthesis tools for FPGA implementation. The library of kernel templates can be further extended when new types of model layer emerge.

We focus on PCI-e based systems for its popularity in data center computing systems. Data communication between *Host* and *Device* are accomplished through a PCI-e slot, and this slot is also used to power on and program FPGA. Inside FPGA, hardware kernels are compiled and invoked by *Host* to perform computation.

IV. Implementation

The great complexity and variability of DNN structures have brought big challenges to generating hardware for each of them individually. It is well-known that DNNs are always constructed by stacking layers. These layers share similar structure in the computation-intensive part, which can always be expressed as or converted into matrix multiplication. As a result, we divide the operations involved in each layer into computation-intensive part and layer-specific part, and implement the computing architecture shown in Figure 3 on FPGA board.

For the computation-intensive part, we use a layer-independent matrix multiplication kernel (*MM*) to perform the calculations. For the layer-specific part, we use a *Data Arranger* to perform data communication with DRAM directly, and it communicates with *MM* through on-chip channels. We store the model configuration file in DRAM. This configuration file includes information of model topological structure, layer specifications, etc. During model inference, *Data Arranger* accesses this configuration file, and parses it to schedule data accessing and kernel execution. In the following subsections, we will present computation-intensive part and layer-specific part respectively, then we will introduce the communication optimizations in detail.

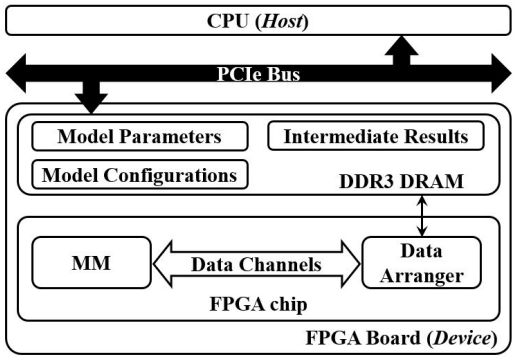


Fig. 3: Overall Architecture of Accelerator

A. Computation-Intensive Part

Considering code efficiency and hardware performance, we implemented *MM* using Verilog. Accelerating matrix multiplication has been a classical problem in the FPGA society, and massive optimizations have been adopted. To better explore the data locality, and make the limited DRAM bandwidth of modern FPGA board match the computing power of *MM*, we take advantage of the tiling strategy to perform matrix multiplication. To insure the multiplication is correctly performed in a tiling manner, we pad zeros to input matrices if any dimension of them is not divisible by its tiling size.

MM takes in two tiles of input matrices, and performs the tiled multiplication vector by vector. All the input data are fed into multipliers simultaneously, then the intermediate results are summed up through a reduction tree to minimize the computing latency. Besides, we use double buffers for the input tiles, and these buffers operate in a ping-pong manner to overlap data communication with computation, which significantly improve the throughput of *MM*.

B. Layer-specific Part

DNNs are constructed by many different types of layers, and the computation and data accessing pattern vary among these layers, so the strategies for converting them into matrix multiplication are also quite different. In the following subsections, we provide details about the operations performed in typical layers, what the computation-intensive part is, and how the *MM* kernel is reused.

1) ***Convolutional Layers****:* Convolutional layers are overwhelmingly popular in applications like image recognition, object detection, object classification, etc. Suppose we have N_{in} input channels and N_{out} output channels. The size of each convolution kernel is $K \times K$, and sliding stride is set to S. The computation during inference phase can be summarized as Equation 1 (bias adding is omitted for simplicity).

$$out[x][y][z] = \sum_{i=1}^{N_{in}} \sum_{j=1}^{K} \sum_{k=1}^{K} (in[i][y \times S + j][z \times S + k] \times W[x][i][j][k]) \quad (1)$$

To perform the computation in convolutional layers with the *MM* kernel, we need to convert the convolution operations

into matrix multiplication. Firstly, we need to turn the input features from a 3-D array into a 2-D array that we can calculate as a matrix. To get a single feature in an output channel, we need to convolve a 3-D cube of input features (also known as a patch) with the corresponding convolution kernels. So we take each one of these input patches and flatten them into a single row of input matrix. This operation is known as *Im2col* (image to column), which is widely applied in prior CPU and GPU studies [7]. With the input features being in a matrix form, we can do similar conversions for convolution kernels by partitioning the corresponding 3-D cubes into a single column of kernel matrix. According to the rules of matrix multiplication, each output channel is serialized into a column of output matrix.

*2) **LSTM Layers**:* In recent years, Long Short-Term Memory (LSTM) has gained great success in Recurrent Neural Network (RNN) design. Numerous variants of LSTM structures have been proposed, while [10] finds that all these variants show little difference in model accuracy. In FP-DNN, we implement the LSTM cell used in [26], which is also supported in TensorFlow. The input of LSTM layer is the combination of input vector at current time-step (in_t) and hidden layer vector at previous time-step (h_{t-1}). Then LSTM layer multiplies input with different weight matrices to get the output vectors of four gates: input gate (I_t), forget gate (F_t), output gate (O_t), and cell gate ($\tilde{C}_t$). Then these output vectors generate the final output vector of LSTM layer with the cell memory of previous time-step (C_{t-1}) through element-wise operations (element-wise addition, multiplication, and activation). The computation performed in LSTM layers is generally shown in Equation 2 to Equation 6, where $sig()$ represents sigmoid function.

$$I_t[x] = sig(\sum_{i=1}^{N_{in}} in_t[i] \times W_{ini}[x][i] + \sum_{i=1}^{N_h} h_{t-1}[i] \times W_{hi}[x][i] + B_i[x]) \quad (2)$$

$$F_t[x] = sig(\sum_{i=1}^{N_{in}} in_t[i] \times W_{inf}[x][i] + \sum_{i=1}^{N_h} h_{t-1}[i] \times W_{hf}[x][i] + B_f[x]) \quad (3)$$

$$\tilde{C}_t[x] = tanh(\sum_{i=1}^{N_{in}} in_t[i] \times W_{inc}[x][i] + \sum_{i=1}^{N_h} h_{t-1}[i] \times W_{hc}[x][i] + B_{\tilde{c}}[x]) \quad (4)$$

$$O_t[x] = sig(\sum_{i=1}^{N_{in}} in_t[i] \times W_{ino}[x][i] + \sum_{i=1}^{N_h} h_{t-1}[i] \times W_{ho}[x][i]] + B_o[x]) \quad (5)$$

$$h_t[x] \ = \ O_t[x] \times tanh(F_t[x] \times C_{t-1}[x] + I_t[x] \times \tilde{C}_t[x]) \quad (6)$$

From the equations above, we can see that the computation-intensive part of LSTM layers is matrix to vector multiplication. Considering vector as a matrix(length at one dimension set to 1), we can map LSTM layer inference to *MM*.

3) Fully-Connected Layers: Fully-Connected layer outputs a vector (out) with input vector (in) and weight matrix (W). Fully-connected layers are also widely deployed in ANNs and classifiers in DNNs. As a result, we design a uniform template for all these layers. The inference phase of fully-connected layers can be summarized as Equation 7. The computation-intensive part is matrix to vector multiplication. Thus, we can re-use *MM* kernel to perform it.

$$out[x] = \sum_{i=1}^{N_{in}} in[i] \ \times \ W[x][i] + B[x] \quad (7)$$

*4) **Other layers**:* Recurrent layer inside simple RNNs (not using LSTM) is actually constructed by adding recurrent connection to fully-connected layer, so the computation-intensive part of both layers is the same. Thus, we can also map recurrent layers to *MM* as we do for fully-connected layers.

Activation layers are always element-wise functions applied to the features, and typical activation functions include $tanh()$, $sigmoid()$, $ReLU()$. Thus, before the layer outputs are offloaded to DRAM, we perform activation functions directly instead of mapping them to *MM*.

Pooling layers extract input features through a sliding window, and choose the average or maximum of this window as output. So there is little computation involved in pooling layers, and there is no need to map them to *MM*. Before outputs are offloaded to DRAM, pooling operations are adopted.

C. Communication Optimization

Since our computation kernels need to communicate with off-chip DRAM for inputs and outputs, the achieved bandwidth is also an important factor to be considered in system design, especially in bandwidth-limited platforms like FPGAs. Previous studies [28] [27] showed that the effective DRAM bandwidth can be raised up by increasing the DRAM burst length. In our on-board test, discontinuous access to DRAM will results in limited burst length, which will degrade the achieved bandwidth to ∼1GB/s. While performing continuous access to DRAM will improve the achieved bandwidth to ∼8GB/s. To prevent I/O from becoming a serious bottleneck of the overall performance, we propose several methods to optimize effective DRAM bandwidth for different layers.

*1) **Convolutional layers**:* For communication optimizations in convolutional layers, we use Figure 4 as a simplified example to illustrate the problems and our solutions. In this example, we set the number of input channels as 8, and each channel has 3×3 elements, so we get 72 input elements in total. The size of convolution kernel is 2×2, and the sliding stride is 1. According to the *Im2col* operations introduced in Section IV.B.1, we can convert the input features into an *Input Matrix*, and we divide this matrix into 4×2 equal tiles. In Figure 4, we show three different layout schemes for comparison: *Im2col*, *Row-major* and *Channel-major*. For each scheme, we show its DRAM layout and DRAM accessing pattern for the first tile.

Im2col: As Figure 4a shows, we can store the entire *Input Matrix* on DRAM by flattening each tile, and insure continuous accessing. However, it is obvious that this scheme stores the whole *Input Matrix* (128 elements) in DRAM, which requires data duplication for adjacent sliding windows. The data duplication brings great overhead on memory footprint, which should be avoided. Besides, to offload the outputs of this convolutional layer as inputs for the following layers, extra operations for data reorganizing and duplication are needed.

Row-major: A straight-forward way to avoid data duplication is: for each channel of *Input Features*, we can store the elements in a row-major manner. We show this scheme in Figure 4b. So this scheme stores 72 elements in DRAM

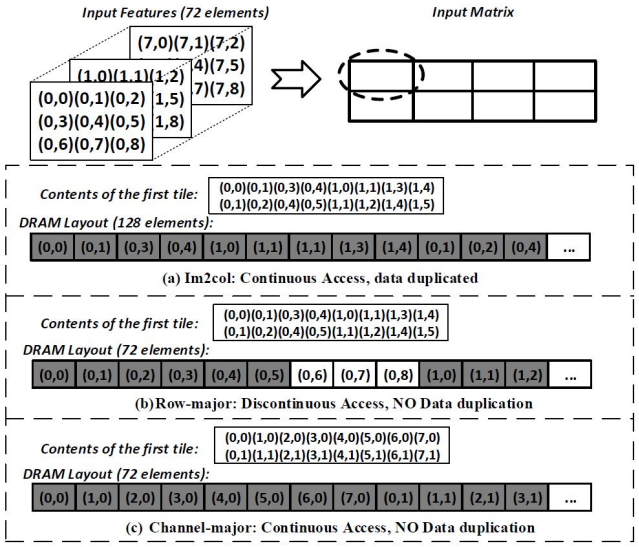


Fig. 4: Layout Optimization

in total, which means there is no data duplication. But we can find that it takes two DRAM bursts to fetch the first tile, which indicates discontinuous DRAM accessing. And this discontinuous accessing pattern will degrade the effective bandwidth. Similar to *Im2col*, offloading the outputs still requires extra operations.

Channel-major: Different from *Row-major*, *Channel-major* stores *Input Features* in a channel-major manner. Thus, *Input Matrix* needs to be reorganized correspondingly: each row (input patch) is also flattened in a channel-major manner. The contents of reorganized first tile is also shown in Figure 4c. So there are in total 72 elements stored in DRAM for *Input Features* without any data duplication. And DRAM is also accessed continuously for fetching input elements. Furthermore, in this scheme, the outputs are also generated in a channel-major manner, which indicates no extra operations for data reorganizing or duplication are needed.

With the comparison above, we choose to use the *Channel-major* scheme to optimize communication of convolutional layers. Along with the *Channel-major* scheme, weight matrix also needs to be adjusted accordingly, but the overhead brought by this can be ignored since weights are pre-trained and these adjustments can be applied before model deployment.

*2) **LSTM Layers & Fully-Conneted Layers**:* According to the algorithm descriptions in Section IV.B.2 and Section IV.B.3, the computation-intensive part of LSTM layers and Fully-connected layers mainly includes matrix to vector multiplication. Unfortunately, the matrix to vector multiplication is inefficient in terms of data locality, because every weight element fetched from DRAM is used only once for a single inference. Thus, most of the inference time are spent on data communication. This indicates that performing model inference directly with *MM* kernel will bring much performance loss. To perform these computations with *MM* efficiently, we propose to batch input vectors together. In this batching way, every element of weight matrices is reused, and we actually convert matrix to vector multiplication into matrix multiplication, which can be efficiently accomplished by the *MM*.

*3) **Other layers**:* The computation-intensive part of recurrent layers is matrix to vector multiplication, which is the same as LSTM layers and fully-connected layers. So we apply similar batching scheme to optimize DRAM communication for them. Other layers like pooling layers and activation layers do not need much data communication with DRAM, so no communication optimization is applied.

D. Data Quantization

Note that numerous prior works [11] [12] have shown that the accuracy of DNNs is robust enough with a decrease in data precision. Many previous works on accelerating DNN inference [23] [19] used fixed-point parameters in their designs for performance improving and resource saving, and this optimization is also called data quantization. So in our implementation, we support implementing fixed-point versions of the target model. Designers using our FP-DNN framework can specify the fixed-point precision by simply using the "-*fixed_point*" compilation option in our *Symbolic Compiler*. In practice, data quantization is done off-line, and the accuracy loss brought by data quantization should be estimated and tested by the users of FP-DNN in advance.

V. Evaluation

A. Experimental setup

In FP-DNN, *Symbolic Compiler* is written in C++ and OpenCL. The HLS code is synthesized by Altera OpenCL Offline Compiler (AOC) [1] (v16.0). HLS-synthesized RTL code is combined with hand-written RTL code and then fed to Quartus 16.0. The code running on the host is written in C++, and compiled with Visual Studio 2013.

For the FPGA platform, we use Catapult [18] system with Altera Stratix-V GSMD5 FPGAs integrated. We use the PikesPeak version of Catapult in our experiments, which has a 4GB DDR3 DRAM as the external memory. The FPGA logic clock frequency is at 150MHz, and the run-time power of the FPGA board is about 25W. This FPGA board is plugged into a PCI-e Gen2 x8 slot of a host computer.

For performance comparison, we use TensorFlow(r0.9) to run model inference on both CPU and GPU. We use a server that includes 2 processors for the CPU implementation, and each processor is a 8-core Xeon E5-2650v2@2.6GHz with a 40MB L3 cache, and the thermal design power (TDP) is 95W. The GPU is an NVIDIA GeForce GTX TITAN X, which has 3072 cuda cores and 12GB GDDR5 memory. The run-time power of it is about 250W. Both CPU- and GPU-implementations run with batch size set to 256.

B. FPGA Resource Utilization

The resource utilization of our *MM* implementations are shown in Table II. Among all the utilized resources, the Catapult Shell (responsible for peripheral interfaces and memory management) and matrix multiplication module are in Verilog,

TABLE I: CNN Performance Comparison with Prior Work

	[23]	[19]	[27]	Our Imp.
FPGA chip	Stratix-V GSD8	Zynq XC7Z045	Virtex-7 690T	Stratix-V GSMD5
Frequency	120 MHz	150 MHz	150MHz	150 MHz
Precision	fixed8-16	fixed16	fixed16	fixed16
DSP Utilization	727/1963	780/900	2833/3600	1036/1590
Overall GOP/S	117.8	137.0	354.0	364.4

TABLE II: Resource Utilization of MM

Precision	float32	fixed16
Logic	164100(95%)	42349(25%)
BRAM	1343(67%)	919(46%)
DSP	264(17%)	1036(65%)

and they take most of the resources. Our *Data Arranger* implemented in OpenCL is very efficient and only takes 2% Logic, 2% BRAM and almost negligible number (8) of DSPs.

C. MM Performance

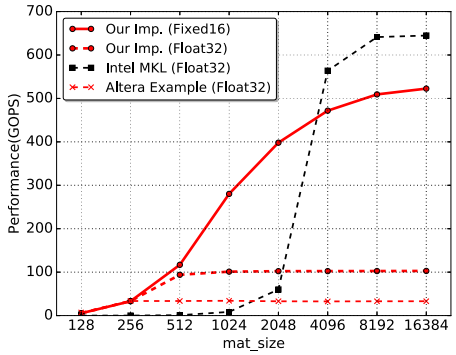
MM is the major build block of our FPGA-based model inference computation, so we compare the performance of our *MM* kernel with other state-of-the-art implementations first. We show the performance (in GOP/S, giga operations per second) of our implementations, Intel MKL [24] and Altera example design [2] for matrix multiplication in Figure 5. Our implementations and Altera example design run on the same FPGA board, and Intel MKL runs on the CPU where all 16 physical cores are fully occupied.

We first evaluate square matrix multiplication in Figure 5a. Among the three implementations on FPGA, our fixed16 implementation achieves the highest performance, and its advantage over the other two implementations accumulates when the matrix size grows. When compared with Intel MKL implementation, our fixed16 implementation runs faster when matrix is small, and MKL only perform better than our implementation when matrix size grows over 4096.

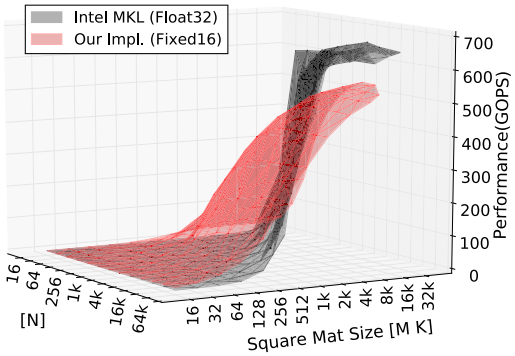
The observation is further confirmed by a wider space exploration of a square matrix to rectangle matrix multiplication in Figure 5b. MKL performs nicely when both matrices are large enough on dimensions, but if the rectangle matrix is very long or very wide, our implementation clearly outperforms MKL, which is usually the case for fully-connected layers and convolutional layers after *Im2col* operations. In another perspective, the MKL performance is achieved when all physical core are fully occupied, which could hinder other tasks from being executed in time.

D. DNN Performance

To show the performance of our FP-DNN framework on a complete model, we compare our CNN implementations with previous accelerators, as shown in Table I. We implement VGG-19 [22], which has 16 convolution layers, 3 fully connected layers and 5 max pooling layers. Since state-of-the-art designs use fixed-point numbers in their implementations, we compare our fixed-point version with them for a fair comparison on performance and resource utilization. The works in [23], [19] and [27] all take the HLS approach (OpenCL-based in [23], C/C++-based in [19] and [27]) for FPGA design. [23]



(a) Square Matrix Multiplication



(b) Square Matrix to Rectangle Matrix Multiplication

Fig. 5: MM Performance Comparison

use an existing matrix multiplication kernel (Altera example design) to perform convolution.[19] design customized convolution kernel for convolutional layers. [27] designed uniformed covolution kernel for both convolutional layers and fully-connected layers. Different from them, our FP-DNN performs convolution with a generalized *MM* kernel, which is designed and optimized under certain hardware constraints. From Table I, we can conclude that the implementations generated by our FP-DNN framework achieve state-of-the-art performance even when compared with hand-coded accelerators which are optimized for certain models. Furthermore, we take the DSP utilization into consideration, and compare our design with the other implementations. It is obvious that our designs perform much better, and use hardware resource more efficiently.

E. Cross-Platform Comparison

To show the great performance and energy efficiency provided by our FP-DNN framework, we compare our implementations with those on CPU and GPU in Table III. We use TensorFlow(r0.9) to run the CPU- and GPU- implementations. We implement several DNNs as benchmarks: VGG-19 [22]

TABLE III: Performance Comparison on Different Platforms

Model	VGG-19[22]			
Platform	CPU	GPU	FPGA	
Pecision	float32	float32	float32	fixed16
Accuracy	89.99%	89.99%	89.99%	89.9%
GOP/S	119	1704	81	364.36
GOP/J	0.63	6.82	3.24	14.57
Model	LSTM-LM[26]			
Platform	CPU	GPU	FPGA	
Pecision	float32	float32	float32	fixed16
Perplexity	78.42	78.42	78.42	78.42
GOP/S	103	1828	86	315.85
GOP/J	0.54	7.31	3.44	12.63
Model	Res-152[13]			
Platform	CPU	GPU	FPGA	
Pecision	float32	float32	float32	fixed16
Accuracy	93.84%	93.84%	93.84%	93.83%
GOP/S	119	1661	73	226.47
GOP/J	0.63	6.60	2.92	9.06

(CNN), LSTM-LM [26] (LSTM- RNN), Res-152 [13] (Residual Net). Performance is evaluated in GOP/S, and energy efficiency is evaluated in GOP/J (giga operations per joule).

We applied data quantization strategies to all three models, and compared the model accuracy between 32-bit floating-point (*float32*) and 16-bit fixed-point (*fixed16*) in Table III. We report the top-5 accuracy of VGG-19 and Res-152 on ImageNet dataset [8]. Higher accuracy indicates the model performs better in the image recognition task. Perplexity of LSTM-LM on PTB dataset [17] is used to evaluate the model. The lower the perplexity is, the better the model performs in the language modeling task. We observe that fixed16 implementations are sufficient for all the networks.

We also compare implementations generated by FP-DNN with other implementations in performance. When we use full-precision (*float32*) data, the implementation generated by FP-DNN is slower than the implementations on CPU. When the data precision is lowered to *fixed16*, FP-DNN implementations are faster than CPU implementations by about 1.9x$\sim$3.06x. We observe that FP-DNN cannot compete with GPU implementations in performance. Regarding energy efficiency, FP-DNN implementations is always better than CPU implementations in all models and precisions. And FP-DNN can easily beat GPU implementations when the data precision is lowered to *fixed16*.

VI. Conclusions

In this paper, we propose FP-DNN, a framework that automatically maps DNNs onto FPGAs to accelerate model inference. FP-DNN analyzes model descriptions to perform model mapping and code generating, then it implements model inference with high-performance computation engine and carefully-designed communication optimization strategies. Our case studies show the great performance and effectiveness achieved by FP-DNN.

VII. Acknowledgement

This work is supported in part by NSF China (No.61572045) and Microsoft Research Asia (No.FY16-RES-THEME-037). We also would like to thank Sixiao Zhu for many inspiring discussions.

References

[1] Altera AOCL. https://www.altera.com/.
[2] Altera OpenCL Example Design. https://www.altera.com/support/support-resources/design-examples/design-software/opencl.html.
[3] CNTK. https://www.cntk.ai/.
[4] Torch. http://torch.ch/.
[5] Martın Abadi, Ashish Agarwal, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.
[6] Frédéric Bastien, Pascal Lamblin, et al. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning, NIPS 2012 Workshop.
[7] Sharan Chetlur, Cliff Woolley, et al. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv: 1410.0759*.
[8] Jia Deng, Wei Dong, et al. ImageNet: A large-scale hierarchical image database. 2009.
[9] Alex Graves, Abdel-rahman Mohamed, et al. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.
[10] Klaus Greff, Rupesh Kumar Srivastava, et al. LSTM: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.
[11] Song Han, Xingyu Liu, et al. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.
[12] Song Han, Huizi Mao, et al. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *CoRR, abs/1510.00149*, 2015.
[13] Kaiming He, Xiangyu Zhang, et al. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.
[14] Yangqing Jia, Evan Shelhamer, et al. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, 2014.
[15] Haoxiang Li, Zhe Lin, et al. A Convolutional Neural Network Cascade for Face Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.
[16] Divya Mahajan, Jongse Park, et al. TABLA: A unified template-based framework for accelerating statistical machine learning. In *IEEE International Symposium on High PERFORMANCE Computer Architecture*, 2016.
[17] Mitchell P. Marcus et al. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 1993.
[18] Andrew Putnam, Adrian M Caulfield, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.
[19] Jiantao Qiu, Jie Wang, et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.
[20] Ruhi Sarikaya, Geoffrey E Hinton, et al. Application of deep belief networks for natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 2014.
[21] Hardik Sharma, Jongse Park, et al. From high-level deep neural models to FPGAs. In *IEEE/ACM International Symposium on Microarchitecture*, 2016.
[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[23] Naveen Suda, Vikas Chandra, et al. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.
[24] Endong Wang, Qing Zhang, et al. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014.
[25] Ying Wang, Jie Xu, et al. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *IEEE/ACM Proceedings of Design Automation Conference*, 2016.
[26] Wojciech Zaremba, Ilya Sutskever, et al. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.
[27] Chen Zhang, Zhenman Fang, et al. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *International Conference on Computer Aided Design*, 2016.
[28] Chen Zhang, Peng Li, et al. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.