An FPGA-Friendly Framework for Designing High-Performance NN Accelerators

Abstract—**Many research efforts have focused on developing automated design flows for efficiently mapping convolutional neural networks (CNNs) to field-programmable gate array (FPGA) platforms in order to generate high-performance accelerators from a high-level definition of a CNN's architecture model. This paper presents F2N2, an end-to-end FPGA-friendly framework for designing high-performance accelerators for CNNs by leveraging the architecture of the target FPGA and the counts and organization of its available resources. We employ a software/hardware co-optimization flow in order to achieve an efficient communication method between the host CPU and FPGA kernel in order to enhance the system-wide speedup. Furthermore, we apply optimizations to reduce the cost of data movement in cloud FPGAs, which are typically equipped with extensive on-chip memory resources. Compared to the state-of-the-art work, F2N2 achieves a factor of three reduction in end-to-end inference latency under the same experimental setup while achieving a clock frequency of 342MHz.**

I. Introduction

Deep neural networks (DNNs) provide state-of-the-art performance in various artificial intelligence applications and have surpassed the accuracy of conventional machine learning models in many challenging domains including computer vision [1]–[3] and natural language processing [4], [5]. Employing deeper models and more advanced models, which have been a main reason for the state-of-the-art performance obtained by DNNs in many domains, have given rise to the extensive computational/memory requirements for deploying DNN models. However, this puts strict design constraints on any domain-specific hardware accelerator design to not only obtain high inference accuracy but also deliver low end-to-end latency, high throughput, and energy efficiency. Convolutional neural networks (CNNs) constitute an important class of DNNs and are mostly used for computer vision tasks. In this context, FPGAs pose as a promising target platform as they can bridge the gap between power-hungry GPU-based accelerators and fixed-function power-efficient ASIC-based designs. The reconfiguration capabilities in FPGAs allow the generation of high-performance, power-efficient hardware accelerator designs that can be configured to meet system-level requirements such as throughput, latency and power in diverse environments, ranging from embedded systems to data centers.

High-level synthesis (HLS) tools have shown substantial progress in generating FPGA-based hardware accelerator designs from a high level of abstraction [6]. Existing HLS tools (e.g., Xilinx's Vivado HLS, Intel's FPGA OpenCL SDK, etc.) employ commonly used programming languages such as C, C++, and OpenCL in order to facilitate the development and design of DNN hardware accelerators. However, they aim to generate an efficient design based on the mapping and scheduling of low-level primitive operations. This leads to a large design space that does not consider the intrinsic structure of the application domain and architecture of the target FPGA device. Therefore, building a high-performance DNN accelerator on an FPGA device is still non-trivial since it requires an effective design space exploration for accelerator design configurations.

Various approaches for automated mapping of DNNs to FPGAs have been presented [7]. Employing these proposed automated tool flows, using DNN-specific algorithmic descriptions and pre-defined hardware templates, an accelerator design implementation can be obtained from a high-level definition (e.g., with Caffe, PyTorch, TensorFlow, etc.). The integration of these automated accelerator generators tool flows in the existing high-level deep learning software frameworks enables the user community to obtain customized hardware implementations of DNNs, without having any hardware design expertise. This enhances the integrability of FPGAs within the deep learning ecosystem. Most existing DNN-to-FPGA tool flows simply employ accelerator designs that are suitable for ASIC implementations also for FPGA realizations, without considering the internal organization of computing and memory resources in the FPGA device. This leads to a architecture-device mismatch which can affect the performance of the accelerator adversely [8]. In addition, these tools flows tend to adequately consider the high cost of data communication between the host CPU and FPGA.

This paper presents F2N2, an end-to-end FPGA-Friendly framework of building and implementing high-performance convolutional Neural Network accelerators on FPGA, leveraging the underlying architecture of the target FPGA device and its available resources. Furthermore, we consider a SW/HW co-optimization flow, achieving an efficient host-FPGA data communication method using techniques such as software pipelining and parallelization. By proposing special optimizations for reducing the data movement costs for the FPGA devices equipped with more extensive on-chip memory resources (cloud FPGAs), we develop low-latency accelerator designs for such FPGA devices. The main contributions of this paper are as follows. (i) We present an end-to-end framework for generating high-performance CNN accelerators suited to a target FPGA device from a PyTorch specification of the network while making highly effective use of available FPGA resources and memory bandwidth. (ii) We employ a SW/HW co-optimization flow, specifically focusing on developing efficient host-FPGA data communication methods by carefully managing and optimizing the host-FPGA interface. (iii) We develop optimized synthesizable C++ templates achieving low-latency accelerator designs on cloud FPGAs, which have large amounts of on-chip memory, by focusing on optimizations that reduce the overhead of data transfers. (iv) We introduce a powerful and flexible compiler (F2N2 Compiler) for mapping a given CNN inference engine running on any dataset onto our optimized accelerator design.

II. Related Work

DNNWeaver [9], fpgaConvNet [10], TGPA [11], cloud-DNN [12], and FlexCNN [13] are some of the prior art references that introduce end-to-end tool flows specifically designed for neural network (NN) acceleration on FPGA devices. fpgaConvNet employs the synchronous dataflow paradigm [14] for mapping and scheduling NNoperations on FPGAs. This is achieved by translating NN operations to a synchronous dataflow hardware graph, applying various transformations on the resulting graph, and solving an optimization problem that maximizes throughput or minimizes latency. fpgaConvNet presents a streaming architecture while the previous work utilize single computation engine. In a streaming architecture, there is one distinct hardware component for each layer and each component is optimized separately, while in single computation engine scheme, a generic accelerator architecture is reused for all layers' computa-

TABLE I: Summary of notation for representing concepts related to compiler for DNN.

Symbol	Meaning
$x/\boldsymbol{x}/\boldsymbol{X}$	input (scalar/vector/two, three-, or four-dimensional tensor)
$y/\boldsymbol{y}/\boldsymbol{Y}$	output (scalar/vector/two, three-, or four-dimensional tensor)
$w/\boldsymbol{w}/\boldsymbol{W}$	weight (scalar/vector/two, three-, or four-dimensional tensor)
L	number of layers
$w_{\mathrm{in}}/h_{\mathrm{in}}$	input width/height
$w_{\mathrm{out}}/h_{\mathrm{out}}$	output width/height
$w_{\mathrm{k}}/h_{\mathrm{k}}$	kernel width/height
$c_{\mathrm{t_{in}}}/c_{\mathrm{t_{out}}}$	number of input/output channels of a tile
$c_{\mathrm{p_{in}}}/c_{\mathrm{p_{out}}}$	number of padded input/output channels to be multiple of $w_{\mathrm{sa}}/h_{\mathrm{sa}}$
$w_{\mathrm{t}}/h_{\mathrm{t}}$	tile width/height
$w_{\mathrm{sa}}/h_{\mathrm{sa}}$	systolic array (SA) width/height
$c_{\mathrm{in}}/c_{\mathrm{out}}$	number of input/output channels
s	stride
p	padding
$w_{\mathrm{p}}/h_{\mathrm{p}}$	width/height of pooling window

tions. TPGA [11] adopts a similar design methodology to improve throughput. Their design aims to optimize throughput by supporting pipelined execution of tiles corresponding to different computation layers for an input image, on multiple heterogeneous accelerators. Additionally, they deliver an end-to-end automation flow to generate the accelerator from the high-level CNN model description.

Frameworks such as cloud-DNN [12] and FlexCNN [13] aim to automate structural optimizations during the FPGA implementation, and creates network description with pre-designed C++ template library. Cloud-DNN targets cloud FPGAs and proposes specialized optimizations with cloud-platform characteristics and the support of easier and streamlined implementation while FlexCNN [13] presents a flow that can called within the TensorFlow framework. FlexCNN also delivers high computation efficiency for different types of convolution layers using techniques including dynamic tiling and data layout optimization.

III. Preliminaries

This section includes background on CNN processing, compiler optimizations, and a description and taxonomy of hardware architectural approaches for designing CNN accelerators.

A. CNN Processing and Notation

Table I summarizes this work's notation in regards to DNNs. The computational flow for a convolutional layer in CNN can be represented by a six-level nested loop (seven-level nested loops by considering the batch size) known as computational block.Recall that a convolutional layer receives input feature maps (IFMs) of size $w_{in} \times h_{in} \times c_{in}$, and convolves them with c_{out} different filters, each filter of size $w_k \times h_k \times c_{in}$, to generate output feature maps (OFMs) of size $w_{out} \times h_{out} \times c_{out}$. The convolution stride for each filter is represented by s. The set of OFMs of the current convolutional layer constitute the IFMs for the next convolutional layer.

B. Compiler Optimizations for CNN Accelerators

Compilers are responsible for performing various optimizations to schedule and map operations defined in NNs onto general-purpose or custom computing processors. Because the accelerator for each layer should perform the same computation, i.e., implement the six-level nested loop, when mapping the convolutional layers of a CNN to a systolic array (SA) of Multiply-Accumulate (MAC) units, the search space for the accelerator may be formally specified by how it transforms (i.e., tiles, reorders, and parallelizes) the nested loop structure for that layer. Although we reuse the same SA for computations of all convolutional layers, each layer has its own unique set of loop transformations. Notice that fully connected layers perform similar computations, but with only a two-level nested loop.

C. Architectural Techniques for Exploiting Data Reuse

Processing DNNs requires performing a large number of MAC operations for calculating outputs of filters/neurons. The Mac operations can be easily parallelized by using spatial architectures include a memory hierarchy, which comprises centralized, large memory blocks in addition to distributed, small memories. While accesses to the large memory blocks incur large latency and come with high energy consumption cost, accesses to small memories are fast and energy-efficient. For large and complex CNN models, it is unlikely that the complete model (including weights) can be mapped onto the chip. Due to the limited off-chip bandwidth, it is critically important to increase the on-chip data reuse and reduce the off-chip data accesses to improve the computing efficiency. Employing different types of computational engines and different designs for the memory hierarchy, we can realize different accelerator designs as will be explained later.

IV. F2N2: Overall Flow

F2N2 provides an end-to-end design framework, which takes a PyTorch description of the CNN and generates a high-performance accelerator suited to the architecture of the target FPGA in three steps (as shown in Fig. 1):

- **Step 1:** F2N2 performs quantization/padding on the test data and pre-trained weights. It may also retrain the quantized network if desired. Next, it generates a inference graph from the given network, as will be discussed in more details in Section VI-A.
- **Step 2:** F2N2 performs optimizations tailored to the employed accelerator design for the implementation of the inference graph. After the optimizations, F2N2 compiles the information from the optimizer and extracts the required parameters for the accelerator design and generates a static schedule. Optimizations and scheduling of operations tailored to the employed accelerator design will be discussed in Section VI-B. Next, it decides on the software optimizations considering the unused resources on target FPGA. The software optimizations are described in Section VII.
- **Step 3:** Finally, using the optimized implemented accelerator component templates, F2N2 generates the hardware code (synthesizable C-level descriptions), that will be used for generating the FPGA bit stream. Main optimizations used in building accelerator component will be discussed in Section V-C.

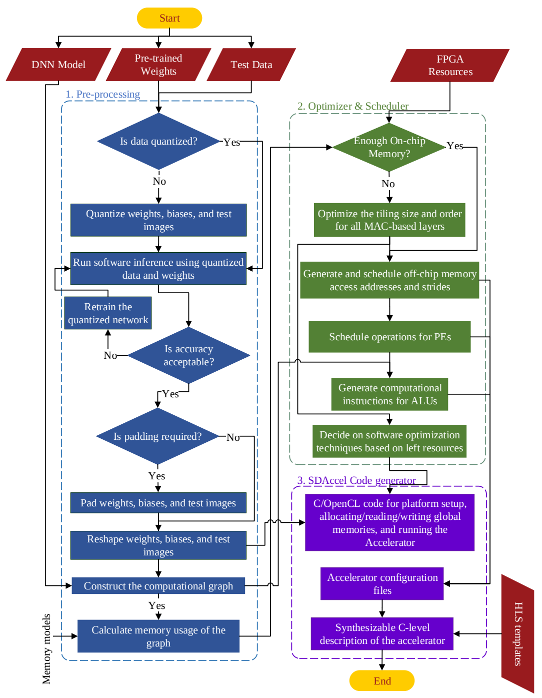
V. Accelerator Design Optimizations

This section focuses on the hardware accelerator design and how to improve its performance using FPGA architecture-aware optimizations, including memory layout and data placement optimizations to reduce the number of accesses to DRAM and increase the effective DRAM bandwidth as well as use of the on-chip memory to balance computation and memory bandwidth.

A. Accelerator Design

The accelerator based on a SA of MAC units typically comprises a 2D array of processing elements (PEs), which are responsible for executing the MAC operations associated with the convolutional operations, and a memory hierarchy feeding data to the said array, compromising of register files, the on-chip memory (Block RAMs and Ultra RAMs on FPGA devices), and external off-chip memory (DRAM). Fig. 2 shows the employed design for the SA of MAC units. As shown, the SA is followed by a processing unit (PU), which in turn comprises several ALUs that are responsible for performing operations such as the nonlinear activation function, pooling, etc.

We present an FGPA architecture-aware accelerator design that utilizes the following structural features of (typical) FPGA devices: (1) In the layout of FPGA, the primitives, e.g., the underlying blocks including digital signal processing units (DSPs), Block RAMs



Start
DNN Model
Pre-trained Weights
Test Data
FPGA Resources
1. Pre-processing
Is data quantized?
Yes
No
Quantize weights, biases, and test images
Run software inference using quantized data and weights
Retrain the quantized network
Is accuracy acceptable?
No
Yes
Is padding required?
No
Yes
Pad weights, biases, and test images
Reshape weights, biases, and test images
Construct the computational graph
Yes
Calculate memory usage of the graph
Memory models
2. Optimizer & Scheduler
Enough On-chip Memory?
Yes
No
Optimize the tiling size and order for all MAC-based layers
Generate and schedule off-chip memory access addresses and strides
Schedule operations for PEs
Generate computational instructions for ALUs
Decide on software optimization techniques based on left resources
3. SDAccel Code generator
C/OpenCL code for platform setup, allocating/reading/writing global memories, and running the Accelerator
Accelerator configuration files
Synthesizable C-level description of the accelerator
HLS templates
End

Fig. 1: A high-level overview of F2N2 flow.

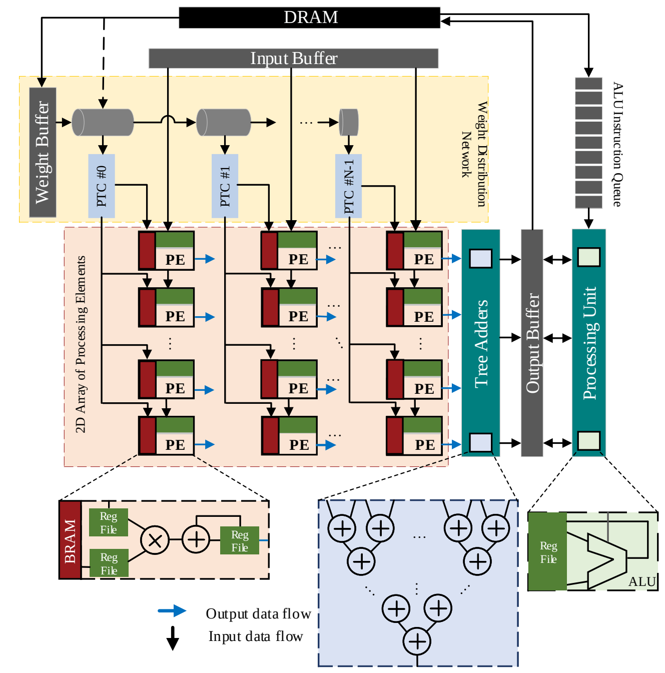


Fig. 2: High-level overview of SA accelerator design. The pipe-shape objects are first-in-first-out buffers (FIFOs) used to distribute the weights. (BRAMs, widely used on-chip memory component in FPGA), optional Ultra RAMs (URAMs, cascadable, single-clocked, two port, synchronous on-chip memory blocks available in UltraScale+ FPGA devices) and Configurable Logic Block (CLBs, comprising several look-up tables, LUTs), are placed column by column. Thus, we have a column of DSPs, followed by a column of CLBs, a column of BRAMs, and a second column of CLBs. On an FPGA device, this resource placement pattern is repeated several times. So, the BRAMs are uniformly distributed on the FPGA chip, and simply connecting a large number of BRAMs to a boundary PE is not wise because it will result in large disparity in routing distances from such a PE to the on-chip memory.(2) The resource ratio of DSP to BRAM is one (a common ratio in most devices), so we can use one BRAM as the storage unit for at least one DSP next to it. In this way, we can reduce the usage of LUTs. This also enables highly parallel implementation of tree adders and ALUs in the PU, for which LUTs are used extensively.

A PE in our design comprises one DSP and its neighboring BRAM. The IFMs are first cached in input buffer and then sequentially passed onto the first row of the SA accelerator. In addition, the input data is shifted into the PE array and between the neighboring PEs so that multiplexers are eliminated. In this way, the global and large fan-out interconnect from input buffers to the all PEs is split into local interconnects between neighboring PEs in rows. After all computations for one OFM are done, the registered partial sum results that reside in the PEs of one row are sent to a (logarithmic) tree adder to do the required summation and produce the final OFM value.

An observation made in [15] confirms that the on-chip data path delay in the SA-based accelerator design goes up when the CNN size increases, but high-level synthesis (HLS) tools do not estimate the interconnect delay correctly or make a conscientious effort to control the quadratic increase of interconnect delay as a function of the interconnect length at this high level of design abstraction. Therefore, we implement pipelined data transfer controllers (PDTCs) for transferring weights, where each PDTC only connects to BRAMs of a set of PEs in a row to reduce the critical path delay. PDTCs are chained together in a linear arrangement using FIFOs. To implement FIFOs, we use LUTRAMs instead of BRAMs. This implementation choice creates a more balanced utilization of the underlying FPGA resources. Each PDTC connects to a local set of buffers to cap the wire delay. The result is an implementation admitting a higher operating frequency with negligible area overhead.

The employed dataflow for the SA accelerator design is a combination of both weight stationery and output stationery dataflows, because (i) all weights required for the computation of the layer are transferred from the external memory into BRAMs and get reused for different patches of input feature maps, and (ii) partial sums computed for generating an OFM are accumulated and stored in the register file of the same PE. More details on the employed dataflow will be discussed in Section VI-B1. Our experiments show that the proposed design can achieve a post-place-and route operating frequency of 344 MHz, when using HLS coding style for mapping to FPGAs. This is the highest frequency reported for an accelerator design implemented in synthesizable C code that we are aware of.

B. Memory Layout and Data Placement

To utilize the off-chip memory bandwidth, we group input/output data as well as weights before sending them to the on-chip memory. Considering weights, we group h_{sa} of them in the output channel dimension. So, given a convolutional layer with a 4D-weight tensor of size $<c_{out}, c_{in}, h_k, w_k>$, we first pad the weights to match the SA dimensions by adding 0's to the c_{out} and c_{in} dimensions. The padded c_{pout} and c_{pin} are multiples of h_{sa} and w_{sa}, respectively. Next, we split the c_{out} dimension into two dimensions with sizes of c_{pout}/h_{sa} and h_{sa}. The final shape of the weight tensor for each layer is then a 5D tensor of $<c_{pout}/h_{sa}, h_k, w_k, c_{pin}, h_{sa}>$, where c_{pout} and c_{pin} are the padded versions of c_{out} and c_{in}, respectively.

Similarly, we pad and group activations alongside the input channel dimension. Since we bring the input data from BRAM and distribute the said data across different PEs within a row of the SA, we group w_{sa} of these data. The output data will be similar to the weights where each h_{sa} of these data are grouped together. Consequently, with a 16-bit fixed point representation for both activation functions and weights, the width of the data stored in output registers and BRAMs for weights and output data of each layer is $h_{sa} \times 16$, whereas the width of data stored in input registers and BRAMs for each layer is $w_{sa} \times 16$. Note that the relative widths of the integer and fractional parts of the fixed-point representation are chosen after training based on the given CNN and the targeted dataset.

C. Low-latency accelerator design

Long latencies associated with access to the external memory constitute a major performance bottleneck for CNNs not only because they have millions of parameters that need to be read from memory, but also because of frequent reading and writing of intermediate partial sum values generated during CNN processing. When we have a target FPGA that has more extensive on-chip memory resources (cloud FPGA), we employ the following optimization techniques in order to decrease the number of read/write accesses from/to the external memory and reduce the time overhead associated with them.

Burst mode transfer: We load all input activations required for a layer's computations at once before any computations for the layer can begin. In this way, we avoid iteratively loading activations from the off-chip memory and writing back calculated partial sums. This lowers the layer's processing time, especially because now loading of weights and input activation functions can be done simultaneously (weights and activations are stored in separate off-chip memory banks in the target FPGA board and are thus simultaneously accessible), and also because we do not need to deal with storing partial sums to the off-chip memory (which in turn reduces both latency and hardware cost). Note that at the conclusion of the layer's processing, the computed output activations are stored to an off-chip memory bank for storing activations.

We also enable the full burst read/write of data from/to the memory banks and utilize the maximum possible burst size (512-bit width) and burst length (256 beats) allowable on the Advanced eXtensible Interface (AXI) bus of the target FPGA board. Specifically, to enable burst read of weights, we allocate a *global weight buffer* on the FPGA device, read all the weights continuously from the off-chip memory bank for weights to this buffer, and then distribute the said weights from the buffer to the distributed BRAM blocks adjacent to the PEs (see Fig. 2). This is more efficient compared to directly reading weights from the memory bank to the distributed BRAM blocks. Similarly, we create global buffers for burst reading (and writing) the input (and output) activations. The said global buffers are realized with the UltraRAM (URAM) resources available on the FPGA device. Employing URAM blocks for implementing global buffers both prevents over-utilizing the BRAM blocks, and helps us achieve a more balanced utilization of resources in the target FPGA board. The advantages of employing efficient utilization of memory hierarchy and burst mode transfer will be discussed in Section VIII-B.

Pre-fetching and double buffering To further reduce the overhead of loading weights, during a layer's computations, we must preferably pre-fetch the weights for the next layer so that the actual computations of the next layer can start earlier. To achieve this goal, we employ two global weight buffers, namely buffer A and buffer B, where one contains the weights required for the current layer's computations while the other is being filled with the weights of the next layer that are read from the off-chip memory. Note that roles of buffers A and B are interchanged for every pair of consecutive layers (this is the notion of double-buffering). Lastly, by using two on-chip activation buffers, and employing the double-buffering technique, we eliminate the need to first store the computed output activations of a layer to the external memory, and then load the the said activations as input activations for the computations of the next layer from the external memory. The benefit of applying the pre-fetching and double buffering optimizations will also be discussed in Section VIII-B.

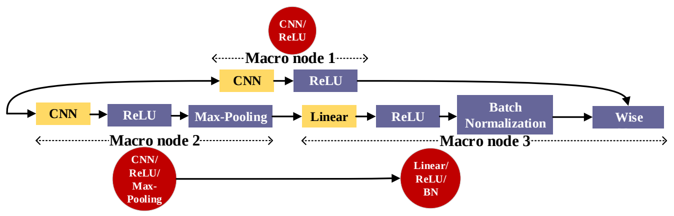


Fig. 3: A given CNN and its corresponding DAG.

VI. F2N2 Compiler

This section describes the proposed NNcompiler which optimizes and runs a NNmodel on the designed accelerator. The proposed compiler takes a NN model, converts it to a computational graph, schedules its operations, and more importantly, optimizes its nodes by leveraging intrinsic fusion of different required operations such as convolution or fully-connected layer computations with batch normalization. Finally, the software optimizations are described in Section VII are used to mitigate the software integration overhead.

A. Pre-processing step and computation graph construction

F2N2 compiler receives a CNN model with pre-trained weights, and test data. It may optionally quantize the weights. Next, it runs the quantized network on the designed accelerator and checks the accuracy. If the accuracy degradation is higher than a user-specified threshold, the compiler invokes a retraining step by using the integrated QPyTorch engine [16] to improve the output accuracy.

A given CNN is a directed acyclic graph (DAG) of various types of layers each performing a required function, e.g., convolution, activation function application, pooling, batch normalization, and flattening (done by the last few fully-connected layers). We use an in-house translator to capture the structure of the input CNN by a DAG where each node in the DAG is a macro node comprising a convolutional or a fully-connected layer and its subsequent processing layer(s) such as activation function and pooling layers up to (but not including) the next convolutional (or fully-connected) layer. Note that the batch normalization is fused into the computations performed by the convolution (or fully-connected) layers later in the proposed flow. Fig. 3 shows a given CNN topology and its corresponding DAG. ResNets incorporate skip connections ("shortcut") between two non-adjacent layers to compute the residual functions and enable more complex interconnections between layers. These characteristics make the ResNet structure highly irregular and more complex compared to other CNNs such as VGGs. To handle such structure, the F2N2 translator first extracts the CNN in the skip connection path as a macro node, and adds an element-wise adder operation as the last macro node of the forward path (see Fig. 3) .

B. Optimizer and Scheduler

As explained in Section V-A, computations are scheduled on the SAs of MAC units of the target FPGA device followed by a PU unit as shown in Fig. 2. More precisely, we use the single computation engine scheme and the MAC-based accelerator design of Fig. 2 for implementing each convolutional or fully-connected layer. However, the compiler can choose a different set of loop optimizations for each layer (including tiling, reordering, and parallelizing the nested loops of computational block).Therefore, the proposed compiler generates custom instructions to determine and distinguish loop alterations and loop bounds for optimizing the performance of each individual layer.

1) Tiling

Our hardware design takes one input image at a time to be sequentially operated on by all nodes in the DAG . After inference on the current image is concluded, the next image is fetched for processing, and so on. For each layer, due to the limited amount of resources that are available on the target FPGA device and in view of much larger volume of required computations and data movements for each layer, our design divides an input feature map into tiles. Next, it loads the tiles one after the other from the off-chip memory to the on-chip memory before processing the said tiles sequentially. We use all the PE units in our SA to process each layer. To partially avoid the resource inefficiency caused by a fixed tile size, our accelerator design allows the use of a dynamic tile size for each DAG node. After extracting the DAG from the given CNN topology, F2N2 compiler calculates the memory requirements for each DAG node. Notice that when the memory requirements are less than the available resources, the compiler does not perform any tiling. Otherwise, it calls its optimizer to apply loop tiling to the loops in the computational block.We make some assumptions for our loop blocking (e.g., tiling) optimization: (1) The compiler does not apply tiling on kernel width and height (w_k and h_k in the computational block) because they are usually small, e.g., 3. (2) The compiler may change the order of the outer loops of Algorithm 1, but maintains the fixed order of the inner loops of Algorithm 2. Note that the order of the inner loops has been chosen so as to minimize data movements considering the architecture and design of the target FPGA device. (3) For the size of the SA ($w_{\mathrm{sa}} \times h_{\mathrm{sa}}$), we search among 16x16, 32x32, and 64x64 size candidates and choose the one that gives us the minimum cost function as explained in Section VI-B2.

Algorithm 1 Tiled computations for a convolutional layer

```
 1: Fill_weight_BRAMs( )
 2: for $m$ in $0 \,..\, \lceil c_{out}/c_{t_{out}} \rceil - 1$ do
 3:     for $y$ in $0 \,..\, \lceil h_{out}/h_t \rceil - 1$ do
 4:         for $x$ in $0 \,..\, \lceil w_{out}/w_t \rceil - 1$ do
 5:             for $n$ in $0 \,..\, \lceil c_{in}/c_{t_{in}} \rceil - 1$ do
 6:                 Load_data( )
 7:                 Do_inner_loops( )
 8:                 Store_data( )
 9:             end for
10:         end for
11:     end for
12: end for
```

Another key point in the tiling optimization step is the management and scheduling of the memory transfer operations of a convolutional (or a linear) layer. As our accelerator design in Fig. 2 illustrates, we bring weights and fill the BRAMs next to each PE before the computational loops are started (cf. $Fill\_weight\_BRAMs()$ on line 1 of Algorithm 1). Input/output feature maps are loaded before the computation engine corresponding to inner loops starts (cf. $Load\_data()$ on line 6 of Algorithm 1), and the generated output feature maps are written back to the main memory (cf. $Store\_data()$ on line 8 of Algorithm 1). The reason that the (partially computed) output feature maps may have to be loaded by $Load\_data()$ is that we also support tiling along the input channel dimension. Indeed, when we tile along this dimension, the partial products that are calculated based on a first subset of the input channels are accumulated and written into the main memory; later when a second subset of the input channels are brought in to continue the pixel value computation for an output feature map, the previously stored pixel value must be read and subsequently combined with the new accumulated value corresponding to the second subset of the input

Algorithm 2 Inner loop computations

```
 1: for $m_t$ in $0 \,..\, \lceil c_{t_{out}}/h_{sa} \rceil - 1$ do
 2:     for $y_t$ in $0 \,..\, h_t - 1$ do
 3:         for $x_t$ in $0 \,..\, w_t - 1$ do
 4:             for $k_y$ in $0 \,..\, h_k - 1$ do
 5:                 for $k_x$ in $0 \,..\, w_k - 1$ do
 6:                     for $n_t$ in $0 \,..\, \lceil c_{t_{in}}/w_{sa} \rceil - 1$ do
 7:                         for $i$ in $0 \,..\, w_{sa} - 1$ do
 8:                             #pragma unroll(i)
 9:                             for $j$ in $0 \,..\, h_{sa} - 1$ do
10:                                 #pragma unroll(j)
11:                                 $\mathbf{O}[(m \times c_{t_{out}} + m_t) \times h_{sa} + j][x \times w_t + x_t][y \times h_t + y_t] += \mathbf{I}[(n \times c_{t_{in}} + n_t) \times w_{sa} + i][x \times w_t + x_t + k_x][y \times h_t + y_t + k_x] \times \mathbf{W}[(n \times c_{t_{in}} + n_t) \times w_{sa} + i][(m \times c_{t_{out}} + m_t) \times h_{sa} + j][k_x][k_y]$
12:                             end for
13:                         end for
14:                     end for
15:                 end for
16:             end for
17:         end for
18:     end for
19: end for
```

channels. Finally, we search over different tile sizes and, based on our computational and memory access models, find the tile size that yields the lowest latency. The computational performance can be calculated as explained in the next subsection. The dataflow used in our design is a combination of weight stationery and output stationery dataflows where the weights are stored in BRAMs and outputs are stored in registers associated with PEs. We also map c_{out} and c_{in} partially to the parallel computation units, i.e., the PEs in the 2-D SA of MAC units. More precisely, sizes of spatial unrolling factors of loops for c_{out} and c_{in} are determined by the the height h_{sa} and width w_{sa} of the SA, respectively.

2) Cost Function

The SA accelerator comprises an array of PEs, an on-chip memory hierarchy, and on-/off-chip interconnects for accessing activation functions and weights. Therefore, we model the performance of the accelerator in terms of computation and communication cost functions, which in turn depend on the tile sizes. The space of all feasible tile sizes can be expressed as:

$$\begin{cases} 1 \le w_t \le w_{out}; \quad 1 \le h_t \le h_{out} \\ 1 \le h_{sa} \le c_{t_{out}} \le c_{out}; \quad 1 \le w_{sa} \le c_{t_{in}} \le c_{in} \end{cases}. \tag{1}$$

The loop ordering in Algorithm 1 is not fixed in advance, and is instead optimized by the the proposed compiler. First, we formulate the computation and data movement costs as follows.

Given a specific tile size combination of w_{t}, h_{t}, $c_{t_{\mathrm{out}}}$, $c_{t_{\mathrm{in}}}$, the computational latency of a convolutional layer may be calculated as:

$$\begin{aligned} T_{comp} = \lceil \frac{c_{out}}{c_{t_{out}}} \rceil \times \lceil \frac{c_{in}}{c_{t_{in}}} \rceil \times \lceil \frac{w_{out}}{w_t} \rceil \times \lceil \frac{h_{out}}{h_t} \rceil \times \\ (h_k \times w_k \times \lceil \frac{c_{t_{out}}}{h_{sa}} \rceil \times \lceil \frac{c_{t_{in}}}{w_{sa}} \rceil \times w_t \times h_t \times u + t_{sa}), \end{aligned} \tag{2}$$

where u denotes the *initiation interval* of the pipeline, which is defined as the number of clock cycles that must elapse between issuing two consecutive input feature maps into the SA ($u = 1$ in our design), and t_{sa} is the (cycle count) latency of passing one input feature map through the SA. Another important factor in the accelerator performance is the data movement (transfer) latency, which is proportional to the total amount of external data accesses from the off-chip memory and thus is calculated as:

$$\begin{aligned} T_{datmov} = DA_{ext} \times t_{ext}, \\ DA_{ext} = \alpha_{in} \times B_{in} + \alpha_w \times B_w + \alpha_{out} \times B_{out}, \end{aligned} \tag{3}$$

where α_{in}, B_{in}, α_w, B_w, α_{out}, and B_{out}, denote the trip counts and buffer sizes of memory accesses to input feature maps, weights, and output feature maps, respectively. t_{ext} denotes the latency of accessing the off-chip memory. Trip count is the total number of transfers made between off-chip memory and buffers. These trip counts and buffers sizes are calculated as follows:

$$\begin{gathered}
\alpha_{in} = \alpha_{out} = \lceil \frac{c_{out}}{c_{t_{out}}} \rceil \times \lceil \frac{c_{in}}{c_{t_{in}}} \rceil \times \lceil \frac{w_{out}}{w_t} \rceil \times \lceil \frac{h_{out}}{h_t} \rceil, \\
\alpha_w = 1, \\
B_{in} = \frac{c_{t_{in}} \times (sh_t + h_k - s) \times (sw_t + w_k - s)}{w_{sa}}, \\
B_w = \frac{c_{out} \times c_{in} \times h_k \times w_k}{h_{sa}}, \quad B_{out} = \frac{c_{t_{out}} \times h_{out} \times w_{out}}{h_{sa}}.
\end{gathered} \tag{4}$$

The reason for dividing all buffer sizes by w_{sa} (h_{sa}) is described in section V-B. α_w is 1 since we bring all the required weights in one external memory trip. By enumerating all possible loop orders and tile sizes, one can generate a set of computational and data transfer latency pairs. The compiler then selects the design with the minimum $T_{datmov} + T_{comp}$. The compiler also has the capability to define the roofline model [17], [18], where the total number of operations (N_{op}), computational roof ($Roof_{comp}$) ratio, and computation to data movement ($C2DM$) ratio are defined as:

$$\begin{gathered}
N_{op} = c_{t_{out}} \times c_{t_{in}} \times h_{out} \times w_{out} \times h_k \times w_k, \\
Roof_{comp} = \frac{N_{op}}{T_{comp}}, C2DM = \frac{N_{op}}{DA_{ext}}.
\end{gathered} \tag{5}$$

TABLE II: Resource utilization and performance of our accelerator

-	Baseline Accelerator	Cloud-targeted Accelerator
Device	Xilinx VU9P on AWS EC2 F1 instance	
Model	VGG-16	
Dataset	CIFAR-10	
Precision	fixed 16-bit	
DSP (%)	15	15
LUT (%)	20	22
BRAM (%)	24	29
URAM (%)	0	33
Frequency (MHz)	342	342
Latency/Image (ms)	22.1	5.05

3) *Generating Instructions*

The proposed compiler generates two sets of instructions, namely, PU compute instructions and SA/tiling configuration instructions for each layer of the CNN as explained below. As defined in section V-A, the PU comprises ALUs where each ALU carries out arithmetic and logic operations on the operands. The compiler takes operations like max pooling and provides a series of ALU and data movement instructions to support such operations. Each of these instructions is 32-bit wide. Configuration instructions are used to dynamically configure tiles for each layer as well as the addresses for data movements, while computational instructions are used in the ALU units to configure the ALU unit to perform the desired operation, e.g., maximum operation for a max pooling layer or addition for an average pooling layer.

ALU instructions are depicted in Table III. the size of immediate operands is 13 bits, which is also adopted for memory, e.g., output buffers and addresses. The next 12 bits are used to represent addresses for the source and destination registers. The 25th bit enables writing back to register files whereas the 26th bit is used to denote whether the second operand is from the second source register file or an immediate. The next three bits are opcode bits that designate the operation to be performed. The last two bits represent the instruction type as to whether the instruction is SA/tiling configuration or ALU compute instructions. Instructions used for configuring the SA are shown in Table III. They are of two types: (i) instructions for addressing DRAM when we load (store) data (results) from (to) DRAM to (from) global buffers; and ii) instructions for setting the accelerator parameters including the tiling data size and counts.

TABLE III: PU operation instruction set

	Bits	31 - 30	29 - 27	26	25	24 - 21	20 - 17	16 - 13	12 - 0
PU-R	**ADD**	**INST-TYPE = 1**	OPCODE = 1	1	1	**DES-ADDR**	**SRC0-ADDR**	**SRC1-ADDR**	-
	SUB		OPCODE = 2	1	1				
	MUL		OPCODE = 3	1	1				
	MVHI		OPCODE = 4	1	1				
	MAX		OPCODE = 5	1	1				
	RSHIFT		OPCODE = 7	1	1				
PU-I	**ADDi**		OPCODE = 1	0	1			-	**Immediate**
	SUBi		OPCODE = 2	0	1				
	MULi		OPCODE = 3	0	1				
	MAXi		OPCODE = 5	0	1				
	RSHIFTi		OPCODE = 7	0	1				
MEM-ALU	**LOAD**		OPCODE = 0	0	1	DES-ADDR	-	-	
	STORE		OPCODE = 6	0	0	-	SRC-ADDR	-	

TABLE IV: Systolic array configuration instruction set

Bits	31 - 30	29 - 28	27 - 0
DRAM-ADDR	INST-TYPE = 0	BUF-TYPE	STARTING-ADDR-POINTER
Bits	31 - 30	29 - 16	15 - 0
CONFIG	INST-TYPE = 0	CONFIG-VAL1	CONFIG-VAL2

4) *Reducing the overhead of batch normalization layers*

The general idea of our approach for reducing the overhead of the batch normalization layer is to fuse the batch normalization layer with the preceding convolutional (or linear) layer into a single layer simply by modifying the layer's parameter values. This simplification also reduces the chances of encountering overflow in our implementations because the parameters of this fused layer will directly generate the normalized results. Algorithm 3 shows the well-known batch normalization algorithm where the two parameters γ and β are learned during the training process. In addition, ϵ is a small constant value used to ensure that division-by-zero error is never encountered.

Algorithm 3 Batch normalization for activation y in a mini-batch, y^{bn} is the normalized result

```
    Input $\mathcal{B} = \{y_0, y_1, ..., y_m\}$; $\gamma$; $\beta$
    Output $y'_i = BN_{\gamma,\beta}(y_i)$
1: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} y_i$                          // mini-batch mean
2: $\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(y_i - \mu_{\mathcal{B}})^2$          // mini-batch variance
3: $\hat{y}_i \leftarrow \frac{y_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$                          // normalize
4: $y'_i \leftarrow \gamma\hat{y}_i + \beta \equiv BN_{\gamma,\beta}(y_i)$                 // scale and shift
```

BN layer after a linear layer: When the BN layer is fused into

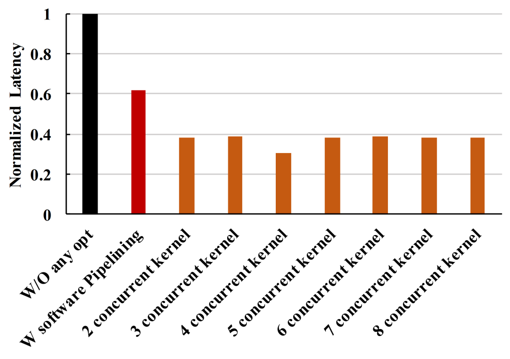


Fig. 4: Normalized latency of running 100 images by considering the proposed software optimizations.

TABLE V: Comparison with Prior Work on VGG-16 *

	F2N2	TGPA [11]	Cloud-DNN [12]	HybridDNN [19]	FlexCNN [13]	DNNWeaver [9]
Device	Xilinx VU9P on AWS EC2 F1 instance			Xilinx VU9P	Xilinx VCU1525	Altera Arria 10 GX115
Model	VGG-16	VGG-19	VGG-16	VGG-16	VGG-16†	VGG-16
Dataset	ImageNet					
Precision	fixed 16-bit			fixed 12-bit	float 32-bit	fixed 16-bit
DSP	4096 (60%)	4096 (60%)	5349 (78.2%)	5163 (75.5%)	3420 (50%)	1344 (88.54%)
LUT	1046452 (88%)	493000 (42%)	764909 (64.7%)	706353 (59.8%)	508363 (43%)	361412 (84.64%)
BRAM	2534 (59%)	3380 (78%)	3456 (80.2%)	3169 (73.4%)	2592 (60%)	2431(89.64%) ‡
URAM (%)	793 (82.60%)	140 (15.6%)	810 (84.4%)	0	144 (15%)	0
Frequency (MHz)	342	210	125	167	242.9	200
Latency/Image (ms)	14.5	22.35	28.96	-	-	-
CNN Performance (GOPS)	2667	1510	1068	3375	-	-

*The values represented with "-" are either not reported or obtainable based on reported results in the corresponding work.
†Their paper has reported results on MobileNetV1, but we ran their open-sourced framework for VGG-16 for our comparison.
‡The width of each BRAM for this Altera FPGA is 20 bit (versus 18 bit in Xilinx boards)

TABLE VI: Comparison with Prior Work on ResNet-18, -50, and -152 networks

	F2N2	**F2N2**	**F2N2**	unzip FPGA [20]	Cloud-DNN [12]	FCNNLIB [21]
Device	Xilinx VU9P on AWS EC2 F1 instance			Xilinx Z7045	Xilinx VU9P	Xilinx VU9P
Model	ResNet-18	ResNet-50	ResNet-152	ResNet-18	ResNet-50	ResNet-152
Dataset	ImageNet					
Precision	fixed 16-bit			fixed 16-bit	fixed 16-bit	fixed 16-bit
DSP	4096 (60%)			900 (100%)	5349 (78.2%)	-
LUT	1046452 (88%)			218600 (100%)	706353 (59.8%)	-
BRAM	2534 (59%)			1092 (12.5%)	3456 (80.2%)	-
URAM (%)	793 (82.60%)			-	810 (84.4%)	-
Frequency (MHz)	342			150	125	200
Latency/Image (ms)	4.43	6.79	14.28	16.7	13.9	14.6
CNN Performance (GOPS)	406	559	791	-	721	1547

a linear layer (one performing a linear transformation on the input features), the fused layer's computation may be described as,

$$y'_j = \gamma_j \frac{(\sum_i w_{ij}x_i + b_j) - \mu_{\mathcal{B}_j}}{\sqrt{\sigma^2_{\mathcal{B}_j} + \epsilon}} + \beta_j \tag{6}$$

Hence, the new fused parameters can be written as:

$$w'_{ij} = \frac{\gamma_j w_{ij}}{\sqrt{\sigma^2_{\mathcal{B}_j} + \epsilon}}, \quad b'_j = \beta_j + \gamma_j \frac{b_j - \mu_{\mathcal{B}_j}}{\sqrt{\sigma^2_{\mathcal{B}_j} + \epsilon}} \tag{7}$$

When the BN layer is fused into a fully connected layer, different input neurons undergo different normalizations i.e., $\boldsymbol{\gamma}, \boldsymbol{\beta}, \boldsymbol{\sigma_{\mathcal{B}}}$ and $\boldsymbol{\mu_{\mathcal{B}}}$ will be vectors which sizes are equal to that of the neuron vector size of the layer (i.e., cardinality of the output vector for the layer).

BN layer after a convolutional layer: With a convolutional layer, the same normalization is applied to all neurons. Using algorithm 3 and proceeding as in the previous case, the weights and bias of the resulting fused convolutional layer may be expressed as:

$$\boldsymbol{w}' = \frac{\boldsymbol{\gamma w}}{\sqrt{\boldsymbol{\sigma}^2_{\mathcal{B}} + \epsilon}}, \quad \boldsymbol{b}' = \boldsymbol{\beta} + \boldsymbol{\gamma} \frac{\boldsymbol{b} - \boldsymbol{\mu_{\mathcal{B}}}}{\sqrt{\boldsymbol{\sigma}^2_{\mathcal{B}} + \epsilon}} \tag{8}$$

VII. SW OPTIMIZATION TECHNIQUES

To realize our design, we use the Xilinx SDAccel and Vivado HLS, which provide a toolchain for programming and optimizing different applications on Xilinx FPGAs using a high-level language (C, C++ or OpenCL) or hardware description languages (VHDL, Verilog and SystemVerilog), and a runtime tool based on the OpenCL API, used by the host-side software to interact with the accelerator.

The optimizations done on the computational kernel (the CNN accelerator) is often offset by the overhead of host-FPGA data communication; this results in only a moderate system-wide speedup or even a slowdown [13], [22]. For example, FlexCNN [13] reports that the time for doing the computations accounts for only 11.8% of the total run-time and the rest of the time is used for data transfers and synchronization. This observation motivates us to develop an efficient host-FPGA data communication method that will maintain the benefits of optimizations that are performed on the computational kernel. Therefore, we use software pipelining and software parallelization to optimize the host-FPGA interface i.e., after doing accelerator kernel optimizations and scheduling the kernel, the F2N2 compiler sets out to optimize the host code. This optimization is make based on the the remaining free resources on the FPGA device as detailed next.

Hiding data transfer time by software pipelining: By default, a kernel can only start processing a new set of data only when it has finished processing the current set of data. By enqueuing data to the device global memory ahead of the kernel execution, data transfer latency can be hidden by such software pipelining. Using OpenCL's clEnqueueMigrateMemObjects command to transfer data, the OpenCL API enables hiding the data transfer time by enqueuing a new set of data while the kernel is operating on the current set. Moreover, by enabling host-to-kernel dataflow, it is possible to further improve the performance of the accelerator by restarting the kernel with a new set of data while the kernel is still processing the previous set of data. The kernel is implemented using the AP_CTRL_CHAIN (Pipelined Kernel) execution model where the kernel is designed in such a way it can allow multiple kernel executions to get overlapped and running in a pipelined fashion. The longer the kernel takes to process a set of data from start to finish, the greater the opportunity to use host-to-kernel dataflow to improve performance.

To run two kernels utilizing the software pipelining technique, the host application starts by sending OpenCL allocated buffers to transfer data from the host code to the global memory. Next the kernel event which is queued in the command queue is notified to start executing the second kernel. Meanwhile the data required for the second kernel execution is transferred using the OpenCL buffers from the host code to the global memory. Thus, once the first kernel execution is completed, the second kernel execution can start immediately and the processed data of the first kernel can be read back from the global memory while the second kernel is executing. F2N2 compiler tries to run as many computational kernels as possible to take advantage of concurrent kernel executions. First, we determine the resource usage of a computational kernel using our estimation models and keep increasing the number of kernels until the available resources are exhausted. The next section explains how concurrent kernel executions is enabled.

Concurrent kernel executions by software parallelization: This allows temporal parallelism, where different stages of the same kernel process different data. This is possible by enqueuing multiple kernels using multiple OpenCL's clEnqueueTask commands in a pipelined manner. clEnqueueTask is used to enqueue a kernel to a command queue. OpenCL's clCreateCommandQueue API creates a command queue that keeps a track of queued tasks. In our design we use an out-of-order command queue to concurrently execute multiple kernels.

Because clEnqueueTask and clEnqueueMigrateMemObject are asynchronous calls to enqueue tasks, event synchronization methods are used to wait and resolve dependencies for all the write events, kernel execution events, and read events. To achieve such synchronization for the host application program the clWaitForEvents API is used to wait for an event and ensure that the task is finished.

VIII. Results and Discussion

A. Experimental Setup

For evaluation purposes, F2N2 targeted a Xilinx VU9P FPGA in the cloud (available on the AWS EC2 F1 instance). This FPGA device includes 64 GiB DDR4 ECC protected memory, with a dedicated PCIe x16 connection. There are four DDR banks. This FPGA contains approximately 2.5 million logic elements and approximately 6,800 Digital Signal Processing (DSP) units. In our implementation, we use three DDR banks, assigning the input feature maps, weights (including bias), and generated instructions, to three separate DDR banks, respectively. Input images are sent using PCIe from the host CPU to the on-board DDR4, accessible by the accelerator, and the output results are sent back to the host CPU. All the architecture choices are parameterizable and can be adjusted based on the target FPGA device. We evaluate F2N2 on well-known CNNs: VGG16 and ResNets and two datasets: CIFAR-10 and ImageNet.

B. Accelerator design optimizations

As mentioned in Section V-C, the timing overhead associated with accessing the off-chip external memory can account for a major portion of the end-to-end inference latency for a given CNN. For instance, to implement the inference computations of VGG-16 with CIFAR-10 dataset using the Accelerator Design Choice #1 and when having a 32x32 SA of PEs, the end-to-end latency is obtained as 22.1 ms, out of which more than 95% is due to timing overheads associated with data transfer from/to the off-chip memory, including the time spent for bringing weights needed for computations of each layer, and the time spent in outer loops for reading/writing tiles of activations from/to the off-chip memory.

Applying the optimizations discussed in Section V-C, simultaneous, burst read of weights and activations significantly improves the computational latency of the layer. Furthermore, because the required weights for the current layer are already loaded into the on-chip global weight buffer (using weight pre-fetching), transferring the weights to the distributed BRAMs starts at the beginning of the layer's computations. Table II reports the frequency and resource utilization of the above configurations. As shown in this table, the end-to-end latency is improved up to 4x compared to the original 22.1ms latency. Furthermore, when employing the accelerator design discussed in Section V-C, employing available URAM resources prevents over-utilizing the BRAM blocks while helping us achieve a more balanced utilization of resources in the target FPGA board.

C. Software optimizations

Next, we study the impact of software optimizations on an end-to-end setup for running a computational kernel as an accelerator. Fig. 4 shows the latency of the running the end-to-end host-accelerator with and without any optimizations. We have normalized the latency with respect to the case without any software optimizations. As Fig. 4 shows, when the software pipelining is enabled, as the read/write latency is overlapped with the computational kernel run, there is significant improvement in the end-to-end latency. The other bars represent the latency when multiple kernels are run concurrently. It can be observed that concurrent kernels reduce the latency significantly. However, by increasing the number of concurrent kernels, the improvement in latency reduces because the latency of data movements for multiple kernels exceeds the kernel execution time.

D. Comparison with State-of-the-Art Designs

Table V shows comparison between F2N2 and prior work references. We use VGG-16 with ImageNet dataset (ImageNet is widely used for evaluation of automated CNN-to-FPGA tool flows). Furthermore, we use our accelerator design discussed in Section V-C, targeting cloud FPGAs. Furthermore, to have meaningful and fair comparisons, we compare our results only with prior works that have used the same or a similar FPGA board as ours. We employ a 64x64 SA of PEs to deal with higher width and height of IFMs for the ImageNet. We also provide comparisons in terms of resource usage (DSPs, LUTs, etc.) on the target FPGA device. From the table, one can see that F2N2 framework used to realize VGG16 inference on ImageNet dataset achieves around 3x end-to-end latency improvement compared to the best latency results reported by state-of-the art work, while using the same (or fewer) number of DSPs and consuming comparable on-chip memory utilization. Table VI shows the obtained performance for F2N2 on ResNet-18 and ResNet-50 networks, alongside a comparison with the state-of-the-art work. As shown in the table, we can achieve up to 4x performance for ResNet-18, and up to 2x performance for ResNet-50 in terms of obtained latency.

IX. Conclusion

In this paper, we presented F2N2, an automated flow for generating CNN accelerators on FPGAs, while making highly effective use of available FPGA resources and memory bandwidth. We also presented a joint SW/HW optimization flow to optimize the host-FPGA interface and hide the latency associated with off-chip memory accesses. Results on VGG16 with ImageNet dataset show a 3X improvement in inference latency compared to recent prior art references.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1106–1114.

[2] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *British Machine Vision Conference*. BMVA Press, 2016.

[3] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2017, pp. 2261–2269.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[6] G. Inggs, S. T. Fleming, D. B. Thomas, and W. Luk, "Is high level synthesis ready for business? A computational finance case study," in *2014 International Conference on Field-Programmable Technology, FPT 2014, Shanghai, China, December 10-12, 2014*, J. Chen, W. Yin, Y. Shibata, L. Wang, H. K. So, and Y. Ma, Eds. IEEE, 2014, pp. 12–19.

[7] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, Jun. 2018.

[8] R. Shi *et al.*, "Ftdl: A tailored fpga-overlay for deep learning with high scalability," in *In Proceedings of ACM/IEEE Design Automation Conference (DAC)*, 2020.

[9] H. Sharma *et al.*, "From high-level deep neural models to fpgas," in *International Symposium on Microarchitecture*. IEEE Computer Society, 2016, pp. 17:1–17:12.

[10] S. I. Venieris and C. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Transaction on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[11] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "TGPA: tile-grained pipeline architecture for low latency CNN inference," in *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, I. Bahar, Ed. ACM, 2018, p. 58.

[12] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-dnn: An open framework for mapping DNN models to cloud fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, K. Bazargan and S. Neuendorffer, Eds. ACM, 2019, pp. 73–82.

[13] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, S. Neuendorffer and L. Shannon, Eds. ACM, 2020, pp. 133–139.

[14] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[15] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Latte: Locality aware transformation for high-level synthesis," in *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 2018, pp. 125–128.

[16] T. Zhang, Z. Lin, G. Yang, and C. D. Sa, "Qpytorch: A low-precision arithmetic simulation framework," *CoRR*, vol. abs/1910.04540, 2019.

[17] L. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, B. L. Hutchings and V. Betz, Eds. ACM, 2013, pp. 29–38.

[18] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[19] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid DNN accelerator design and implementation," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6.

[20] S. I. Venieris, J. Fernández-Marqués, and N. D. Lane, "unzipfpga: Enhancing fpga-based CNN engines with on-the-fly weights generation," *CoRR*, vol. abs/2103.05600, 2021. [Online]. Available: https://arxiv.org/abs/2103.05600

[21] Q. Xiao, L. Lu, J. Xie, and Y. Liang, "Fcnnlib: An efficient and flexible convolution algorithm library on fpgas," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/DAC18072.2020.9218748

[22] J. Cong, P. Wei, and C. H. Yu, "From JVM to FPGA: bridging abstraction hierarchy via optimized deep pipelining," in *10th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2018, Boston, MA, USA, July 9, 2018*, G. Ananthanarayanan and I. Gupta, Eds. USENIX Association, 2018.