Inf4Edge: Automatic Resource-aware Generation of Energy-efficient CNN Inference Accelerator for Edge Embedded FPGAs

Ali Jahanshahi*, Rasool Sharifi†, Mohammadreza Rezvani*, Hadi Zamani*
*Department of Computer Science and Engineering, University of California, Riverside. Riverside, CA, USA
†Department of Computer Science, University of Virginia, Charlottesville, VA, USA
ajaha004@ucr.edu, as3mx@virginia.edu, mrezv002@ucr.edu, hzama001@ucr.edu

***Abstract*—Convolutional Neural Networks (CNN) have achieved great success in a large number of applications and have been among the most powerful and widely used techniques in computer vision. CNN inference is very computation-intensive which makes it difficult to be integrated into resource-constrained embedded devices such as smart phones, smart glasses, and robots. Along side inference latency, energy-efficiency is also of great importance when it comes to embedded devices with limited computational, storage, and energy resources. Embedded FPGAs, as a fast and energy-efficient solution, are one of widely used platforms for accelerating CNN inference. However, the difficulty of programming and their limited hardware resources have made them a less attractive option to the users.**

In this paper, we propose Inf4Edge, an automated framework for designing CNN inference accelerator on small embedded FPGAs. The proposed framework seamlessly generates a CNNs inference accelerator that fits the target FPGA using different resource-aware optimization techniques. We eliminate the overhead of transferring the data to/from FPGA back and forth which introduces latency and energy consumption. To avoid the data transfer overhead, we keep all of the data on the FPGA on-chip memory which makes the generated inference accelerator faster and more energy-efficient.

Given a high-level description of the CNN and a data set, the framework builds and trains the model, and generates an optimized CNN inference accelerator for the target FPGA. As a case study, we use 16-bit fixed-point data in the generated CNN inference accelerator on a small FPGA and compare it to the same software model running on the FPGA's ARM processor. Using 16-bit fixed-point data type results in $\sim 2\%$ accuracy loss in the CNN inference accelerator. In return, we get up to $15.86\times$ speedup performing inference on the FPGA.

***Index Terms*—CNN inference, Energy-Efficiency, Embedded FPGA, Accelerator, Edge Computing**

I. Introduction

With the rapid advances in computational power, Convolutional Neural Networks (CNN) have demonstrated the potential to surpass human-level performance in a variety of fields including robotics [1], natural language processing (NLP) [2], biosensing [3], hardware design and optimization [4], and especially in computer vision applications [5]. CNN models and architectures are continuing to get more complicated and larger to achieve higher performance. Higher performance of CNNs comes at the cost of significant computational and memory resources demand which makes the deployment of CNNs on resource-constrained devices a challenge.

As CNNs inference applications are becoming ubiquitous, the need for moving the processing power to edge devices has been increasing [6]. Latency and uncertainty of sending (receiving) the data through a unreliable network to (from) cloud where powerful servers perform the computations is the main motivation for moving processing power to edge devices. When it comes to designing CNNs to run on edge devices, most of the time the application is latency-sensitive and real-time which makes the CNN accelerator latency a critical factor [6]. In addition to real-time nature of CNN applications on the edge devices, due to battery or temperature restrictions, most of edge devices have a very limited energy/power budget [7], [8]. Therefore, the edge devices are designed with minimal computation and storage resources, which as a result, makes it very difficult to reuse already designed and implemented applications for them.

Application Specific Integrated Circuits (ASIC), Field Programmable Gate Arrays (FPGA), and Graphical Processing Units (GPU) are among the most popular and practical CNN inference accelerator solutions being widely used on edge devices. Designed/programmed for a specific application, as we move from ASIC to FPGA and from FPGA to GPU, overall design cost and latency decrease, and power/energy consumption and reprogrammability (flexibility) increase. GPUs provide tens of GOP/sec, which results in high CNN inference throughput. Although due to power hungry nature of GPUs, researchers have been trying to make them more energy-efficient for different applications by several microarchitectural tweaks [9], [10] as well as system level optimizations [11]–[13], they are not a good solution for edge devices with strict power/energy constraints. Furthermore, works like [14], [15] have demonstrated limitations of using GPUs for CNN workloads. On the other hand, ASICs' are designed and produced for a specific CNN, making them very fast and power-/energy-efficient, but they are not programmable. Therefore, FPGAs are placed between ASICs and GPUs when it comes to power/energy consumption, flexibility, and latency.

A large body of research has been conducted on developing accelerators specifically customized for FPGA architecture for different applications such as database sort operations [16] and AI-based game acceleration [17]. CNN inference acceleration has also been popular among researchers in the past several

years. Networks with binary weights [18], [19] have shown accuracy comparable to full precision nets. Binarized neural networks (BNNs) as an efficient solution for developing CNNs on FPGA reduce storage and memory bandwidth requirements and replace floating point operations with binary operations. BNNs due to their especial architecture have a lot of implementation challenges. Therefore, we do not consider BNN approaches in this work. In this work, in order to address the memory bandwidth challenges [20], [21], we try to minimize the memory transfer overhead by fitting the weights in the FPGA on-chip memory.

In addition, designing hardware accelerators for CNN inference on FPGAs requires significant investment and time to develop. However, the ML/DL community constantly keeps to develop new complicated CNN models or improve existing ones for better accuracy and performance which makes designing new FPGA implementations or upgrading the existing solutions to recent proposed models. Therefore, showing the necessity of having a framework that can automate the CNN inference accelerator design and development on FPGAs. High-level synthesis (HLS) tools such as Xilinx Vivado HLS [22] and LegUp [23] enable designers to write code in a high-level programming language (such as C/C++), then the tool compiles it to a register-transfer level (RTL) design specification. More recent tools such as Intel FPGA SDK for OpenCL [24] and Xilinx SDSoC [25] offer further automation features for generating the hardware-software interface and on-chip memory network. The existing HLS tools and frameworks target general algorithms which is not very optimized when it comes to a very specific applications, i.e. CNN inference accelerator for very small embedded FPGAs in our case. Overlay processors such as Xilinx Deep Learning Processor Unit (DPU) allow deployment of CNNs on a user-programmable dedicated engine by translating a high level description of the model [26]. However, these tools have shown to suffer from shortcomings such as limited function support and extra configuration time overhead [27]

This work proposes an automated resource-aware framework for CNN inference accelerators on small embedded FPGAs. Our framework automatically generates an optimized hardware for a CNN architecture. The framework consists of three main components that correspond to our contributions:

- **Hardware Generation Pipeline:** a pipeline to generate the hardware for a given high-level description of a CNN. This pipeline build the model from the description, generates the HDL for the CNN inference accelerator, and synthesizes the generated HDL to produce the bitstream for the target FPGA. In Section II-A we go through details of this pipeline.
- **Precision Adjustment Loop:** we target very small embedded FPGAs in this work so we use fixed-point data in the generated accelerator. Using a model-in-the-loop concept, we adjust the precision of the fixed-point data produced by each layer of the CNN to reduce impact of the quantization error propagation in the network. This loop will be explained in Section II-B

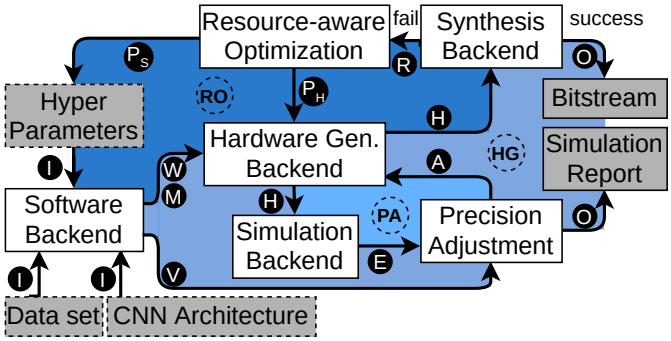

Fig. 1: Inf4Edge framework. Grey boxes with dashed borders illustrate the framework inputs provided by the designer as a high-level JSON description. Grey boxes with solid borders are the framework outputs, i.e., bitstream and generated hardware accuracy report. Stages to produce the output are shown by circles with dashed border, where PA is Precision Adjustment, HG is Hardware Generation, and RO is Resource-aware Optimization.

- **Resource-aware Optimization Loop:** in case the generated CNN inference accelerator does not fit into the target FPGA, this optimization loop first, trades speed for less hardware resource usage, and then trades a little model accuracy drop for less hardware resource usage. We will elaborate on this loop in Section II-C.

The rest of this paper is organized as follows; we first describe the framework's overall structure, then go into details for each component of the framework. Finally, we provide the results discussing different aspects of the generated CNN for our target FPGA.

II. Inf4Edge Framework

Figure 1 shows the overall flow of the inference for edge (Inf4Edge) framework. There are different pipelines by which for a given input ❶ a CNN inference hardware is generated and is optimized for the target embedded FPGA. *Hardware Generation* pipeline (HG) generates the CNN inference accelerator HDL ❽. Hardware generation is resource agnostic which means it does not take the target FPGA hardware resources into account. However, it provides different knobs to trade the accelerator's inference latency for hardware resources.

We use fixed-point data types in the hardware. Therefore, we need to accurately adjust the precision of the intermediate data in the hardware. *Precision Adjustment* stage (PA) fine-tunes the data precision between different CNN accelerator layers. In case the generated hardware does not fit in the target FPGA, *Resource-aware Optimization* stage (RO) minimizes the CNN hardware resource usage with minimum accuracy degradation.

In the following subsections, each stage of the framework is explained in details.

A. Hardware Generation (HG)

Hardware generation includes a pipeline of procedures performed to generate the bitstream to be programmed on the target embedded FPGA from a given CNN high-level

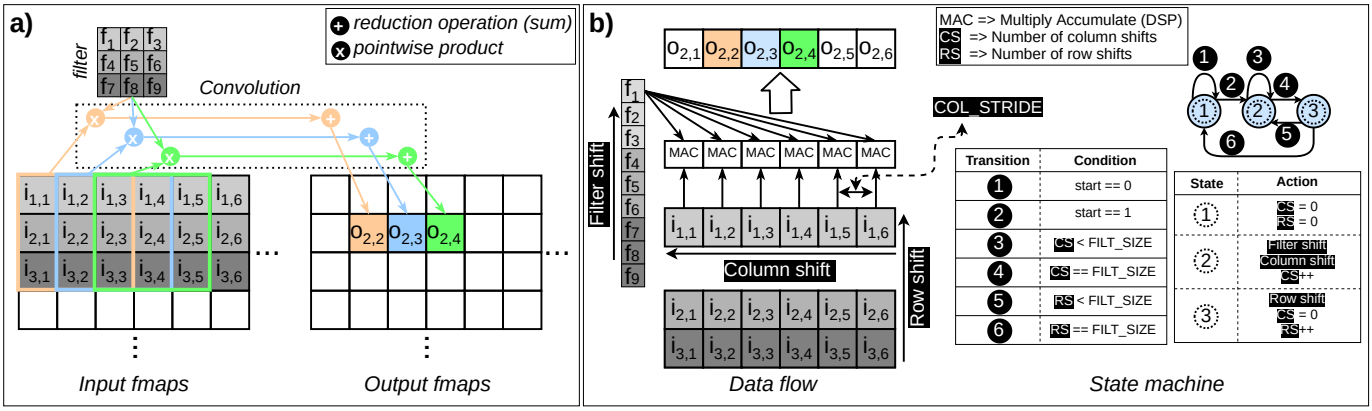

Fig. 2: a) Convolution data dependencies. b) Designed convolution unit's Data Flow and State Machine for $FILT\_SIZE = 3$ and $COL\_STRIDE = 1$. The *DSP*s are wired to input elements based on COL_STRIDE which is 1 in here.

specification. Hardware generation pipeline does not include any optimization; However, different knobs are provided by different components of this pipeline to control resource usage of the generated hardware. This pipeline is shown by (HG) in Figure 1. In the following subsections, different components of this pipeline are explained.

1) ***Software Backend****:* the software backend provides an API that accepts a JSON file containing high-level specifications of the CNN, including its architecture (number and type of the layers, their connections), hyperparameters, and the data set. Abstracting the CNN specifications eliminates the need for designers to deal with the programming part of the CNN design. As a result, the design process will be faster and less error-prone. This backend performs two main operations, which are essential for the next steps:

1.1) CNN designing, training, and model generation: given a high-level CNN specifications by the designer, the software backend first splits the data set into training, validation, and test data sets. Then, based on the CNN architecture, the software model of the CNN is built using *Python* library [28]. The software backend uses several techniques (Dropout, regularization, and data augmentation) to prevent model over-fitting [29], [30].

After training the CNN using the provided data set, the weights of the most accurate model are quantized to 16-bit fixed-point data. The quantized weights (W), as well as the CNN model architecture (M), will be passed to the hardware generation backend to produce the hardware for the model and to initialize the weights into the FPGA BRAMs.

1.2) Verification (inter-layer) data: after training the model, the software backend feeds a portion of random samples from test data to the trained model and saves the output of all layers of the CNN model for each sample. We call the collected data as verification (inter-layer) data denoted by (V) in Figure 1. Verification data are passed to precision adjustment stage (PA) for precision adjustment and verification of difference accelerator's components.

2) ***Hardware Generation Backend****:* this backend generates the hardware description (H) of the CNN model (M) generated by software backend. We have implemented configurable templates of all of the widely-used hardware components required to implement CNNs on FPGAs. This backend is developed in *CHISEL*, a powerful, well-known hardware construction language that enables the designers to implement advanced hardware components using highly parameterized generators [31].

The implemented hardware components templates are *modular* and *parameterized*. The components can be configured based on the CNN specifications, including data bit-width, shared or exclusive components (convolution unit), and the number of *DSP*s used for components (convolution and fully connected units). Furthermore, all of the implemented hardware components are modular, which means we can put them together with few lines of code, similar to designing CNNs in a deep learning software framework. Since hardware generation backend is a part of several pipeline/loops, the generate hardware components parameters are set at this backend based on inputs from two other framework stages:

1) The model specification inputted to this stage by software backend (M) (Section II-A1).
2) Hardware optimization (P_H) stage (Section II-C1).

The generation of the CNN accelerator based on the model architecture is automated in the hardware generation backend, which is completely transparent from the designer. The following subsections contain the design and implementation details of each hardware component used in generating CNN accelerator:

2.1) Convolution unit: convolution operation consists of several multiplies and accumulate (MAC) operations. In order to design a highly parallel convolution unit, we need to find the pattern by which the data is being accessed for consecutive convolution operations. Figure 2.a illustrates the access pattern for three convolution operations with a filter with size $FILT\_SIZE = 3$ and stride size $COL\_STRIDE = 1$. Based on Figure 2.a, in order to compute each element of the output, we need $FILT\_SIZE$ rows and $FILT\_SIZE$ columns of the input *fmap*. A filter is applied on a $FILT\_SIZE \times FILT\_SIZE$ region of the input *fmap* and pointwise product is computed. The results of all of the multiply operations are added together

by a summation reduction operation. Equation 1 shows the data access for one single convolution to produce an output element of output *fmap* ($o_{r,c}$).

$$\begin{aligned} o_{r,c} = i_{r-1,c-1} * f_1 \;&+ i_{r-1,c} * f_2 \;+ i_{r-1,c+1} * f_3 \\ + i_{r,c-1} * f_4 \;&+ i_{r,c} * f_5 \;+ i_{r,c+1} * f_6 \\ + i_{r+1,c-1} * f_7 \;&+ i_{r+1,c} * f_8 \;+ i_{r+1,c+1} * f_9 \end{aligned} \quad (1)$$

To compute the convolution of the next output element of the same row ($o_{r,c+1}$), the same filter is shifted by one on the input *fmap*, the convolution is performed, and after the reduction operation, the result will a single scalar value. Different window colors in Figure 2.a show this. Therefore, in order to produce one row of the output *fmap*, the same filter is slid and applied to $FILT\_SIZE$ rows of the input *fmap* resulting in reusing one element of each row $FILT\_SIZE$ times. We leveraged this access pattern in the data flow design of the convolution unit.

Figure 2.b illustrates the data flow and state machine of the generated convolution unit by the framework. To compute the convolution of each row of the output *fmap*, $FILT\_SIZE = 3$ rows of the input *fmap* are fed to this unit alongside the filter to be convolved with the input.

As shown in the data flow part of Figure 2.b, the filter gets flattened in the generated convolution unit data flow, and its first element is wired to one input of all DSPs (MAC) units. One row of the input *fmap* is also wired to their corresponding DSP based on the COL_STRIDE value. In Figure 2, COL_STRIDE is equal to 1, so consecutive input elements are wired to the neighboring *DSP*s. In case COL_STRIDE is set to 2 by the designer, every other input element would be wired to the neighboring *DSP*s.

As illustrated in Figure 2.b., the generated state machine for the convolution unit performs $FILT\_SIZE = 3$ shifts on the input columns and filter, then it shifts rows by 1. This process is performed $FILT\_SIZE$ times to compute the convolution for $FILT\_SIZE$ rows. The output of this hardware unit is one row with the size equal to "input *fmap* width divided by COL_STRIDE". The output goes through an activation layer, a max-pooling layer, and a precision adjustment layer to get to the next FeedForward layer.

We use built-in FPGA *DSP*s for MAC operations. A challenge of using the *DSP*s (compared to other resources) is that they are very limited in quantity. Therefore, the implemented convolution unit hardware is parameterized so that the number of *DSP*s for this unit can be specified, and the hardware generation backend generates the convolution unit's state machine in a way that it uses only the specified number of *DSP*s. The more *DSP*s we allocate to this hardware unit, the higher throughput we get. The maximum number of *DSP*s is the input picture's width (input *fmap* of the first convolution layer), and the minimum is one. For the sake of simplicity, in Figure 2, the illustrated data flow and the state machine use *DSP*s equal to the input picture's width. This parameter will be fine-tuned by the resource-aware optimization stage, which will be explained later.

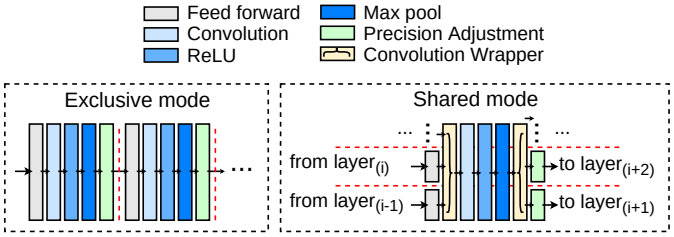

Fig. 3: Exclusive vs. Shared mode for convolution unit. Red dashed lines specify the CNN convolutional layers.

Furthermore, the framework allows selecting from two hardware generation modes for this unit; exclusive mode and shared mode.

- **Exclusive mode:** as illustrated in Figure 3, choosing the exclusive mode, for each convolution layer in the designed CNN model, a convolution unit hardware is generated. This mode of hardware generation results in higher throughput and higher FPGA resource usage. This mode is enabled by default in the framework.
- **Shared mode:** using this mode, the convolution unit is shared among all layers of CNN. Sharing this unit results in using less FPGA hardware resources, but it introduces throughput degradation. Figure 3 illustrates this mode. By enabling this mode, a wrapper is generated automatically by the hardware generation backend. The wrapper acts as a resource access manager for this unit by arbitrating different layers requesting the shared convolution unit. Since the input elements are wired to the *DSP*s in the convolution unit, for the layers with the same COL_STRIDEs, one convolution unit has to be generated and shared. Therefore, the number of shared convolution units in the generated CNN hardware depends on the number of different COL_STRIDEs in the CNN, which is specified by the hyper parameters ❶.

The accuracy of the generated CNN is independent of either of these modes. The provided generation modes enable the framework to trade latency/throughput for hardware resource usage for very small embedded FPGAs. Similar to the number of allotted *DSP*s for the convolution unit, the resource-aware optimization stage determines the exclusive or shared generation mode.

2.2) Activation and Max pooling units: in the proposed framework, we implemented a max-pooling unit and an activation hardware unit. The activation unit performs a Rectified Linear (*ReLU*) function. The max-pooling unit is parameterized and is generated to perform $MP\_SIZE \times MP\_SIZE$ max-pooling where MP_SIZE is the size of max pooling window (a hyper parameter).

2.3) Inter-layer precision adjustment unit: As mentioned before, we are using 16-bit fixed-point data in our framework. Using fixed-point instead of floating-point introduces quantization error to the layers' output. This error propagates through the network and affects the final results adversely, resulting in accuracy degradation. Since the size of the integer/fraction part of the 16-bit fixed-point data propagated through the network varies from one layer to another, we need to adjust the size of

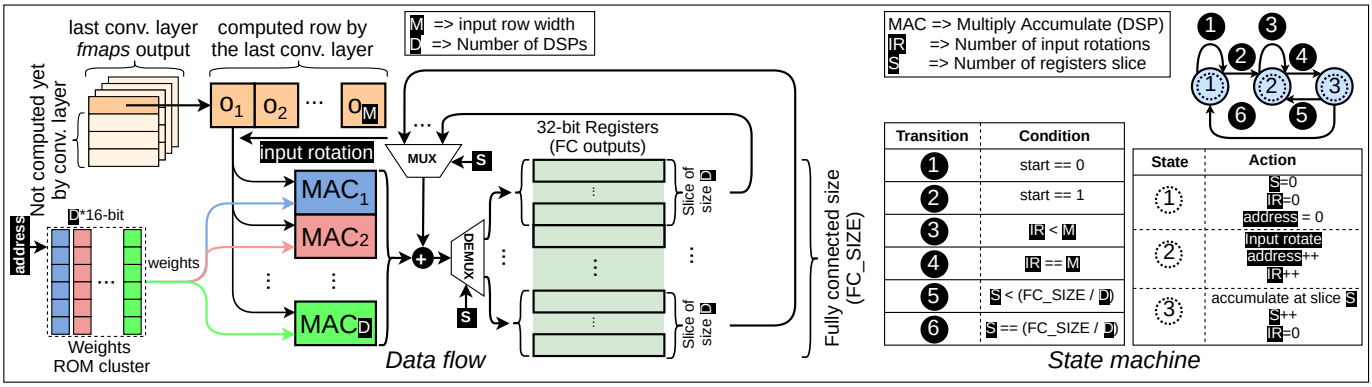

Fig. 4: Fully Connected (FC) unit Data flow and State machine. FC_SIZE represents the number of neurons (output size) in the FC unit, and D represents the number of DSPs used for this unit regardless of input and output sizes. The outputs of this layer are 32-bit to preserve accuracy.

the integer part and fraction part at the end of each layer of the network as illustrated in Figure 3. For doing so, we insert an Inter-layer precision adjustment unit at the end of each layer. This unit is responsible for adjusting the integer and fractional part of the output data of each layer before it is fed to the next layer of the network. The adjustment of this unit is performed by Precision Adjustment (PA) stage (Section II-B).

2.4) FeedForward unit: this unit is responsible for buffering and performing accumulation of the input *fmaps* in a buffer (BRAM). It is also responsible for feeding the previous layer's accumulated *fmaps* to the convolution unit ($FILT\_SIZE$ rows at a time with ROW_STRIDE strides) in conjunction with the *filter* that is going to be convolved with the *fmap*. Figure 5 shows the structure of this hardware unit for $FILT\_SIZE = 3$ and $ROW\_STRIDE = 1$.

The buffering state machine is generated specific to each layer since the dimension of inter-layer data for each layer is different (due to max-pooling and $COL\_STIRDE > 1$). The BRAM and ROM in Figure 5 are generated according to the size of the *fmaps* that will be buffered in the layer and the number of the layer filters. ROMs are initialized with the quantized CNN weights (W) provided by the software backend.

On the output side, the feeding state machine loads the rows from the *fmaps* RAM based on ROW_STRIDE & $FILT\_SIZE$ and feeds them into the output with the filter.

The generated buffering and feeding SMs by hardware generation backend are decoupled in order to avoid stalls to the pipeline, which diminishes the throughput.

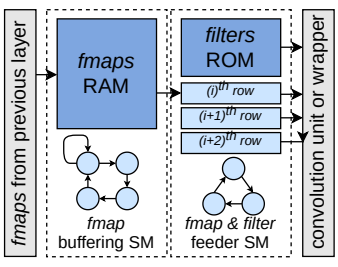

Fig. 5: FeedForward unit architecture with $FILT\_SIZE = 3$ & $ROW\_STRIDE = 1$. *fmap* buffering and feeder state machines (SM) are decoupled to avoid the CNN pipeline stalls, resulting in higher throughput.

2.5) Dense or fully-connected (FC) unit: dense or fully-connected is also implemented as a parameterized hardware unit. Similar to the convolution unit, this unit mainly consists of multiplication and addition (MAC), requiring FPGA's built-in *DSP*s. In this framework, the number of DSPs (D) and the number of neurons in the FC unit (FC_SIZE), which corresponds to the FC unit output size, are configurable.

In most CNNs, the FC unit size (the number of neurons) is quite large. However, due to FPGA resource constraints, we may not be able to use *DSP*s equal to the size of the FC unit. Therefore, this unit is generated in a way that is able to handle the difference between the number of used *DSP*s (D) and FC unit output size (FC_SIZE) for this unit under the conditions stated in Equation 2.

$$1 \leq D \leq FC\_SIZE$$
$$FC\_SIZE \,\%\, D = 0 \tag{2}$$

Figure 4 shows the data flow and the state machine for this unit. The state machine performs partial MAC operations (in slices of size D) and accumulates the results in the output registers of the FC unit. For the first slice of size D ($S = 0$), the state machine rotates the input by one to the left at each cycle, resulting in feeding the different elements of the input to be MACed by their corresponding weights stored in ROMs. After M cycles (size of the input row), the result of partial MAC is accumulated in the FC output registers, MAC units are reset, and the same input row gets MACed by the weights to generate the next slice of size D ($S = 1$). This process of accumulation is performed $S = FC\_SIZE \,/\, D$ times so that all of the input row elements are MACed by all of the neurons in the FC unit.

The weights for this layer are stored in a cluster of ROMs where each ROM contains the weights for its corresponding *DSP* or MAC unit. The ROMs are generated and initialized based on the weights provided by the software backend and are used in the synthesis.

Figure 4 illustrates the state machine for processing only one input *fmap* row. We design the FC unit state machine in a way that the unit does not wait for all of the *fmaps* to be

produced by the last convolution layer. This unit processes the last convolution layer output row after it is outputted by the convolution layer to keep the pipeline full.

In conjunction with the state machine shown in Figure 4, which process only one input row, another state machine is generated for the FC unit to keep track of all of the input *fmap*s as the convolution unit outputs them. At the end of each inference stage, the higher-level state machine outputs the result of FC operation, which is accumulated in 32-bit registers, and resets them to make the FC unit ready for the next inference.

In our experiments, we noticed that this unit could be the bottleneck of the generated CNN accelerator. Depending on the number of used *DSP*s and the FC unit's size, this unit's latency for processing one input row can be higher than that of convolution unit. If the convolution unit is used in exclusive mode, the rate at which the convolved rows are fed into the FC unit is higher than the rate at which the FC unit processes them. This situation worsens when the target FPGA is so tiny that the allocated *DSP*s for FC unit is very low compared to the size of FC unit, i.e., high FC_SIZE to D ratio. In order to address this problem, we put a buffer between the last convolution unit and the FC unit. Using a shared convolution unit does not cause these problems since the convolution unit becomes the bottleneck of the CNN.

3) ***Synthesis Backend****:* this backend synthesizes the generated hardware ❽ by hardware generation backend for the target FPGA. If the synthesis is successful, a bitstream is generated to program the FPGA with. In case of synthesis failure, the report of synthesis ❾ is fed to resource-aware optimization as a guide to further optimize the hardware for the target FPGA.

B. Precision Adjustment (PA)

One important step in generating the CNN is to adjust the precision of each layer output before feeding them to the next layer. In order to adjust the fixed-point precision of the CNN, we need to propagate several test inputs to the hardware implementation of the CNN. The precision adjustment process leverages the model-in-the-loop concept. This stage is demonstrated in Figure 1 by Ⓟⓐ.

As we mentioned before in Section II-A1, verification (inter-layer) data ❺ are generated by the software backend. In this stage, an accurate software model of the generated CNN hardware is used in simulation to verify the correctness of the design and to perform inter-layer precision adjustment. For doing so, all of the verification data are fed to the model in simulation. Then, the output of each layer of the generated CNN hardware model ❽ is collected, and the difference of the collected data and the verification data are calculated to measure the error ❹ introduced by the precision adjustment in each layer. Starting from the first layer, this step configures the inter-layer precision adjustment unit ❶ so that the output error of the first layer is minimized. Then, the output error of the next layer is measured, and the precision of the inter-layer precision adjustment unit for that layer is set.

The sequence of propagating data, collecting data, calculating error, and adjusting the precision units is performed until the generated hardware's inter-layer and accelerator's end-to-end error merges to a minimum error. i.e., as close as possible to verification (inter-layer) data. In the end, the simulation and precision adjustment backend reports the error introduced by quantization of the model to 16-bits as well as the accuracy of the generated hardware for the CNN ❿.

The precision adjustment step leverages *CHISEL*'s powerful test capability to input data to the design hardware and analyzes it. We use Verilator [32], a cycle-accurate hardware simulation engine, as the simulation backend of this stage in our *CHISEL*-based framework. Verilator converts the HDL description of the hardware design ❽ to a C++ or SystemC model that can be compiled and executed. Since the model is converted to a C++ executable, the simulation is performed fast.

C. Resource-aware Optimization (RO)

The proposed framework targets very small embedded FPGAs. Therefore, one important step for designing the CNN is to check if the target FPGA has enough computational (LUT and DSP) and memory (Flip Flop and BRAM). The goal of this stage is to use the existing knobs to optimized the resource usages of the CNN so that it fits on the target FPGA with minimum impact on the model performance, i.e., accuracy. If the synthesis of the generated hardware accelerator fails (due to a lack of enough hardware resource for the design), this stage tries to optimize the hardware to fit it in the target FPGA. This stage consists of two steps that are performed in order.

1) ***Hardware Components Optimization****:* Hardware components optimization consists of a model-in-the-loop adjustment of the generated hardware parameters without affecting the accuracy of the model. At this step, the framework trades latency/throughput of the accelerator for its hardware resource usage by fine-tuning different components of the generated accelerated to fit it into the FPGA. For doing so, the report of the synthesis backend ❾ is analyzed, and depending on the type of resource (LUT/FF, BRAM, DSP) that caused the synthesis to fail, the hardware components parameters P_H are adjusted. Algorithm 1 shows how this stage is performed.

At this stage, the framework cannot provide any solution to the memory (BRAM) shortage by adjusting the parameters of the hardware components. Since the model weights ❼ are stored in BRAMs of the FPGA, in case the synthesis failure is due to a lack of enough BRAMs, the framework goes back to the inputs (hyper paramters) to optimize the CNN software model size (fewer parameters) in the software model optimization step.

2) ***Software Model Optimization****:* This step aims to decrease the number of the model parameters so that the new model fits in the target FPGA. At this step, the framework trades model accuracy for memory.

Bayesian optimization has previously been used for auto-tuning high-level synthesis directives [33], and extracting

Algorithm 1: Harware Components Optimization

Data: Ⓗ: hardware description of CNN
Ⓡ ←Synthesize(Ⓗ).report
while Ⓡ.$failed$ **do**
  P_H ← Ⓗ.$conf$
  if $failure\ is\ due\ to\ lack\ of\ LUT$ **then**
    P_H ← P_H.Conv.gen_mode = Shared
    Ⓗ = HardwareGen(conf)
  else
    if $failure\ is\ due\ to\ lack\ of\ DSP$ **then**
      P_H ← $adjustNumDSPs$(P_H.FC)
      P_H ← $adjustNumDSPs$(P_H.Conv)
      Ⓗ = HardwareGen(conf)
    else
      if $failure\ is\ due\ to\ lack\ of\ BRAM$ **then**
        SoftwareModelOptimization(Ⓡ)
        break
  Ⓡ ←Synthesize(Ⓗ).report

accelerator variants with optimal performance and cost trade-offs on the FPGA [34]. In order to converge to the desired smaller model, Bayesian Optimization is used in the proposed framework alongside some heuristic to adjust the hyper parameters of the model P_S so that the accuracy degradation of the new model is minimum while the model is smaller.

III. Experimental Results

We use Inf4Edge framework to generate a CNN for images classification on a small embedded FPGA. Our experiment setup detail is as follows:

CNN Architecture: Table I shows the architecture and hyper parameters of the CNN we input to the framework in order to classify CIFAR-10 dataset images.

Dataset and preprocessing: we use CIFAR-10 dataset to train and test the CNN. 85% of the data were used for training and validation, and 15% of the data were used for test.

Hardware Platform (FPGA): we evaluate our design on a *PYNQ* board [35] which is an open-source project from *Xilinx* that makes it easy to design embedded systems with *Xilinx Zynq SoC*. *PYNQ* uses a low-cost *Xilinx Zynq-7000 SoC* containing an *XC7Z020* FPGA alongside an *ARM Cortex-A9* embedded processor.

Synthesis Tool: we use *Xilinx SDSoC 2016.4* to perform synthesis, generate reports used in resource-aware optimization stage, generate bitstream, and programming the FPGA.

Resource-aware optimization: we intentionally set $FILT\_SIZE = 5$, and $D = FC\_SIZE$ in framework's input hyper parameters. Since this hyper parameter results in a model which is larger than our FPGA capacity, the synthesis fails. First, the framework set the the convolution unit generation mode to *shared* mode in the hardware components optimization step, but it did not help and the synthesis failed again. In the next effort, hardware component optimization picked $D = 50$ and $D = 10$ for the first and the second FC units respectively, which means instead of using $D_1 = FC\_SIZE_1 = 100$ and $D_2 = FC\_SIZE_2 = 10$ *DSP*s for the FC units, 50 and 10 were used respectively.

TABLE I: The CNN architecture generated by the framework.

Layer	Filt. size	Output size	# of output *fmaps*	# of params
Conv2d	3	(32, 32)	32	320
ReLU		(32, 32)	32	0
Max pool		(16, 16)	32	0
Conv2d	3	(16, 16)	64	18496
ReLU		(16, 16)	64	0
Max pool		(8, 8)	64	0
Conv2d	3	(8, 8)	128	73856
ReLU		(8, 8)	128	0
Max pool		(4, 4)	128	0
Conv2d	3	(4, 4)	128	147584
ReLU		(4, 4)	128	0
Max pool		(2, 2)	128	0
Dense		(1, 100)		51300
ReLU		(1, 100)		0
Dense		(1, 10)		1010
ReLU		(1, 10)		0
Total				292,566

In the third round of optimization, the FPGA could not fit the model weights in the BRAMs. Therefore, software model optimization step decided to use $FILT\_SIZE = 3$ as filter size for all of the convolution layers. The reason behind choosing such a small CNN is that we are using a very small FPGA.

Accuracy loss, HW vs SW: in order to compare the accuracy of the software model and the accelerator implemented on the FPGA, we classified CIFAR-10 test dataset with both software model on *ARM Cortex-A9* which is the embedded processor of our FPGA and on the CNN accelerator with two modes.

Table II shows the accuracy. *SW* is the *Python*-based software classifier performed on ARM processor of the FPGA board (650 MHz frequency). *HW-SM* is the CNN accelerator generated in shared mode (sharing one convolution unit among all layers), and *HW-EM* is the CNN accelerator generated in exclusive mode (one convolution for each layer).

TABLE II: Software vs. Inf4Edge inference accelerator.

version	Accuracy (%)	Runtime (ms)	Data type
SW	74.13	51.24	32-b floating
HW-SM	72.04	11.17	16-b fixed
HW-EM	72.04	3.23	16-b fixed

We see that the effect of using 16-bit fixed-point data on the precision of the classification is negligible which is congruent with previous work on the effect of quantization on the CNNs' accuracy. Also, as we expected, using exclusive convolution hardware units improves the runtime by avoiding stalls due to serializing the use of convolution unit which is shared among 4 layers.

Figure 6 shows the resource usage of the CNN hardware implemented in *PYNQ* board. As we expected, memory (BRAM) is the hardware resource limitation that we encounter since BRAMs have to be utilized for storing model and all inter-layer data.

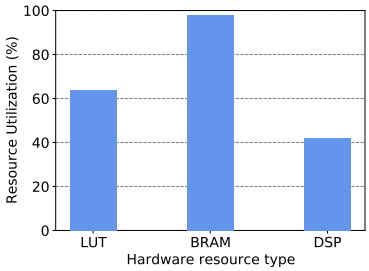

Fig. 6: Hardware resource utilization of the Table I CNN implemented on FPGA.

IV. Conclusion

We proposed a framework that enables the designers to automatically implement a CNN inference accelerator for an embedded FPGA without having the knowledge of hardware design. The framework provides a software API that allows designers to input a high-level description of their desired CNN design to the framework, and the framework generates the CNN accelerator for the target FPGA considering its hardware resource limitation. The resource-aware optimization first tries to fine-tune the parameters of the hardware, and if it still fails, the software model hyper parameters are fine-tuned for the target FPGA so that it fits into the FPGA. The framework also tunes the precision of the the intermediate data using the data set provided by the designer.

References

[1] S. Kumra and C. Kanan, "Robotic grasp detection using deep convolutional neural networks," 2017.

[2] W. Wang and J. Gang, "Application of convolutional neural network in natural language processing," in *International Conference on Information Systems and Computer Aided Education (ICISCAE), 2018*.

[3] H. Raji, M. Tayyab, J. Sui, S. R. Mahmoodi, and M. Javanmard, "Biosensors and machine learning for enhanced detection, stratification, and classification of cells: A review," 2021.

[4] A. Fouman Ajirlou and I. Partin-Vaisband, "A machine learning pipeline stage for adaptive frequency adjustment," *IEEE Transactions on Computers*, 2021.

[5] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *CoRR*, vol. abs/1901.06032, 2019.

[6] L. Du, Y. Du, Y. Li, J. Su, Y.-C. Kuan, C.-C. Liu, and M.-C. F. Chang, "A reconfigurable streaming deep convolutional neural network accelerator for internet of things," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2018.

[7] Z. Abbas and W. Yoon, "A survey on energy conserving mechanisms for the internet of things: Wireless networking aspects," *Sensors*, 2015.

[8] S. R. M. Hassan Raji, Pengfei Xie and M. Javanmard, "Wireless power-up and readout of label-free electronic detection of protein biomarkers," in *25th International Conference on Miniaturized Systems for Chemistry and Life Sciences, 2021*.

[9] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO), 2021*.

[10] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, 2020*.

[11] H. Zamani, Y. Liu, D. Tripathy, L. Bhuyan, and Z. Chen, "Greenmm: energy efficient gpu matrix multiplication through undervolting," in *ACM International Conference on Supercomputing, 2019*.

[12] H. Zamani, D. Tripathy, and et al., "Saou: Safe adaptive overclocking and undervolting for energy-efficient gpu computing," in *IEEE International Symposium on Low Power Electronics and Design, 2020*.

[13] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers," *IEEE Computer Architecture Letters, 2020*.

[14] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 128–131, 2019.

[15] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021.

[16] B. Romanous, M. Rezvani, and et al., "High-performance parallel radix sort on fpga," in *2020 IEEE 28th International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020*.

[17] A. Jahanshahi, M. K. Taram, and N. Eskandari, "Blokus duo game on fpga," in *The 17th CSI IEEE International Symposium on Computer Architecture & Digital Systems (CADS), 2013*.

[18] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[19] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, 2016.

[20] R. Sharifi and Z. Navabi, "Online profiling for cluster-specific variable rate refreshing in high-density dram systems," in *ETS*, 2017.

[21] S. S. N. Larimi, B. Salami, O. S. Unsal, A. C. Kestelman, H. Sarbazi-Azad, and O. Mutlu, "Understanding power consumption and reliability of high-bandwidth memory with voltage underscaling," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021*.

[22] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *Transactions on Embedded Computing Systems (TECS)*, 2013.

[24] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 2012.

[25] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 4–4.

[26] Xilinx, "DPU for convolutional neural network v3.0, DPU IP product guide," 2019.

[27] A. Kalantar, Z. Zimmerman, and P. Brisk, "FA-LAMP: fpga-accelerated learned approximate matrix profile for time series similarity prediction," in *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021*.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[29] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[30] L. Xie, J. Wang, Z. Wei, M. Wang, and Q. Tian, "Disturblabel: Regularizing cnn on the loss layer," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4753–4762.

[31] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC)*. IEEE, 2012.

[32] W. Snyder, "Verilator: Open simulation-growing up," *DVClub*, 2013.

[33] A. Mehrabi and et al., "Prospector: Synthesizing efficient accelerators via statistical learning," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020*.

[34] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Bayesian optimization for efficient accelerator synthesis," *ACM Transactions on Architecture and Code Optimization (TACO), 2020*.

[35] "Pynq: Python productivity for zynq," http://www.pynq.io/, accessed: 04/21/2019.