Design Exploration Framework for Floating-Point CNN Acceleration on Low-Power Resource-Limited Embedded FPGAs

1st
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—In this article, we present a design exploration framework for floating-point convolutional neural networks (CNNs) acceleration on low-power, resource-limited embedded FPGAs targeting IoT sensor data analytic applications. We propose a scalable hardware architecture with customizable tensor processors (TPs) integrated with TensorFlow Lite. The implemented hardware optimization realizes hybrid custom floating-point and logarithmic dot-product approximation. This approach accelerates computation, reduces energy consumption and resource utilization while maintaining inference accuracy. Experimental results on MiniZed (XC7Z007S) and Zybo (XC7Z010) demonstrate peak acceleration and power efficiency of 105X and 5.5 GFLOP/s/W, respectively.**

***Index Terms*—Artificial intelligence, convolutional neural networks, depthwise separable convolution, hardware accelerator, TensorFlow Lite, embedded systems, FPGA, custom floating-point, logarithmic computation, approximate computing**

I. Introduction

THE constant research and the rapid evolution of machine learning (ML) techniques for sensor data analytics represent a promising landscape for Internet-of-Things (IoT) endpoint applications. CNN-based models represent the essential building blocks in 2D pattern recognition tasks. Sensor-based applications such as mechanical fault diagnosis [1], [2], structural health monitoring (SHM) [3], human activity recognition (HAR) [4], hazardous gas detection [5] have been powered by CNN-based models in industry and academia.

Due to the high computational demands of CNNs, dedicated hardware is typically required to accelerate execution. In terms of computational throughput, graphics processing units (GPUs) offer the highest performance. In terms of power efficiency, ASIC and FPGA solutions are well known to be more energy efficient (than GPUs) [6]. As a result, numerous commercial ASIC and FPGA accelerators have been proposed, targeting both high performance computing (HPC) for data-centers and embedded systems applications [7], [8].

However, most of these CNN accelerators have been implemented to target mid- to high-range FPGAs to compute intensive CNN models such as AlexNet, VGG-16, ResNet-18. The power supply demands, physical dimensions, air cooling and heat sink requirements, and in some cases their elevated costs make these implementations inadequate or even impossible on resource-constrained low-power IoT devices.

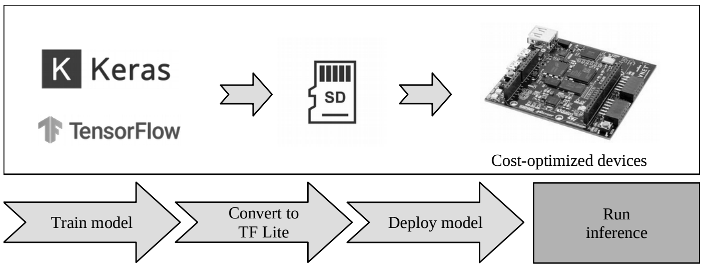


Fig. 1. The workflow of our approach on embedded FPGAs.

In this article, we propose a design exploration framework for floating-point custom shallow CNN acceleration targeting low-power, inexpensive embedded FPGAs. This framework integrates TensorFlow (TF) Lite library with delegate interfaces between software runtime and the proposed hardware architecture to accelerate *Conv2D* and *DepthwiseConv2D* tensor operations. We design a tensor processor (TP) as a low-power hardware engine with customizable resource utilization. To accelerate floating-point computation, we adopt the hybrid custom floating-point and logarithmic dot-product approximation technique [9], which exploits the intrinsic error-resilience of neural networks [10]. Further on, we propose a quantize aware training method to maintain and increase inference accuracy with low-precision custom floating-point formats.

To operate the proposed system, the user would train a custom CNN model using TensorFlow or Keras, then this is converted into a TensorFlow Lite model, finally, the model is stored in a micro SD card along with the embedded software and configuration bitstream. (See **Fig.** 1.)

Our main contributions are as follows:

1) We develop a hardware/software co-design framework targeting low-power, resource-limited embedded FPGAs for floating-point CNN acceleration. This is a scalable and parameterized architecture integrated with TensorFlow Lite that allows hardware design exploration.
2) We present a customizable tensor processor (TP) as a

dedicated hardware accelerator. This TP computes *Conv2D* and *DepthwiseConv2D* tensor operations employing hybrid custom floating-point formats and hybrid logarithmic approximation with parametrized on-chip memory utilization.

3) We propose a quantize aware training method that maintains and increases inference accuracy with low-precision custom floating-point formats.
4) We demonstrate the potential of the proposed architecture by addressing a design exploration with custom shallow CNN models using *Conv2D* and *DepthwiseConv2D* tensor operations. We evaluate compute performance and classification accuracy.

The rest of the paper is organized as follows. Section II covers the related work; Section III introduces the background to *Conv2D* and *DepthwiseConv2D* tensor operations; Section IV describes the system design of the hardware/software architecture and the quantized aware training method; Section V presents the experimental results thorough a design exploration flow; Section VI concludes the paper.

To promote the research in this field, our entire work is made available to the public as an open-source project at (*hidden for double blinded review*).

II. Related work

A. Google's Edge TPU

The Edge Tensor Processing Unit (TPU) is an ASIC designed by Google that provides high performance machine learning (ML) inference for TensorFlow Lite models [11]. This implementation uses PCIe and I2C/GPIO to interface with an iMX 8M system-on-chip (SoC). The reported throughput and power efficiency are 4 TOPS and 2 TOPS per watt, respectively [12]. The Edge TPU supports 40 tensor operations including *Conv2D* and *DepthwiseConv2D*.

However, the Edge TPU does not support floating-point computation. The Edge TPU supports only TensorFlow Lite models that are 8-bit quantized and then compiled specifically for the Edge TPU [13]. Regarding power dissipation, the Edge TPU system-on-module (SoM) requires up to $15W$ power supply [12], which can be inadequate for very low-power applications.

B. Xilinx Zynq DPU

The Xilinx deep learning processing unit (DPU) is a configurable computation engine optimized for CNNs. The degree of parallelism utilized in the engine is a design parameter and can be selected according to the target device and application. The DPU IP can be implemented in the programmable logic (PL) of the selected Zynq-7000 SoC or Zynq UltraScale+ MPSoC device with direct connections to the processing system (PS) [14]. The peak theoretical performance reported on Zynq-7020 is 230 GOP/s.

However, the DPU does not support floating-point computation. The DPU requires the CNN model to be quantized, calibrated, converted into a deployable model, and then compiled into the executable format [14].

III. Background

A. Conv2D tensor operation

The *Conv2D* tensor operation is described in **Eq.** (1), where h is the input feature map, W is the convolution kernel (known as filter), and b is the bias for the output feature map [15]. We denote *Conv* as *Conv2D* operator.

$$Conv\left(W,h\right)_{i,j,o} = \sum_{k,l,m}^{K,L,M} h_{(i+k,j+l,m)} W_{(o,k,l,m)} + b_o \tag{1}$$

B. DepthwiseConv2D tensor operation

The *DepthwiseConv2D* tensor operation is described in **Eq.** (2), where h is the input feature map, W is the convolution kernel (known as filter), and b is the bias for the output feature map. We denote *DConv* as *DepthwiseConv2D* operator.

$$DConv\left(W,h\right)_{i,j,n} = \sum_{k,l}^{K,L} h_{(i+k,j+l,n)} W_{(k,l,n)} + b_n \tag{2}$$

IV. System Design

In this section we describe the system design as a hardware/software co-design framework for floating-point CNN acceleration targeting resource-limited FPGAs. This is a scalable and parameterized architecture that allows design exploration integrated with TensorFlow Lite.

A. *Base embedded system architecture*

As a hardware/software co-design, the system architecture is an embedded CPU+FPGA-based platform, where the acceleration of tensor operations is based on asynchronous[1] execution in parallel TPs. **Fig.** 2 illustrates the system hardware architecture as a scalable structure. For operational configuration, each TP uses AXI-Lite interface. For data transfer, each TP uses AXI-Stream interfaces via Direct Memory Access (DMA) allowing data movement with high transfer rate. Each TP asserts an interrupt flag once the job or transaction is complete. Interrupt events are handled by the embedded CPU to collect results and start a new transaction.

The hardware architecture can resize its resource utilization by modifying the number of TP instances prior to the hardware synthesis, this provides scalability with a good trade-off between area and throughput.

B. *Tensor processor*

The TP is a dedicated hardware module to compute tensor operations. The hardware architecture is described in **Fig.** 3. This architecture implements high performance off-chip communication with AXI-Stream, direct CPU communication with AXI-Lite, and on-chip storage utilizing BRAM. This hardware architecture is implemented with HLS. The tensor operations are implemented based on the C++ TensorFlow Lite micro kernels.

[1]The system is synchronous at the circuit level, but the execution is asynchronous in terms of jobs.

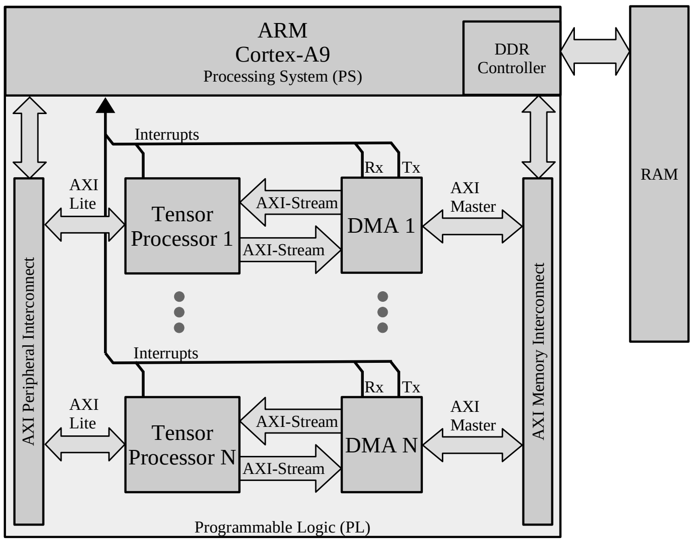


Fig. 2. Base embedded system architecture.

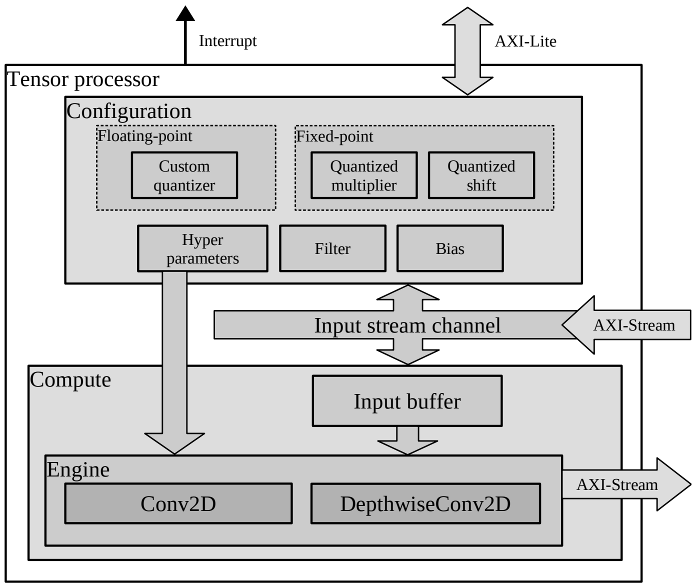


Fig. 3. Hardware architecture of the proposed tensor processor.

*1) **Modes of operation**:* This accelerator offers two modes of operation: *configuration* and *execution*.

- In *configuration* mode, the TP receives the tensor operation ID and hyperparameters: stride, dilation, padding, offset, activation, depth-multiplier, input shape, filter shape, bias shape, and output shape. Afterwards, the TP receives filter and bias tensors to be locally stored.
- In *execution* mode, the TP executes the tensor operator according to the hyperparameters given in the configuration mode. During execution, the input and output tensor-buffers are moved from/to the TF Lite memory regions via DMA.

*2) **Dot-product with with hybrid custom floating-point and logarithmic dot-product approximation**:* We optimize the

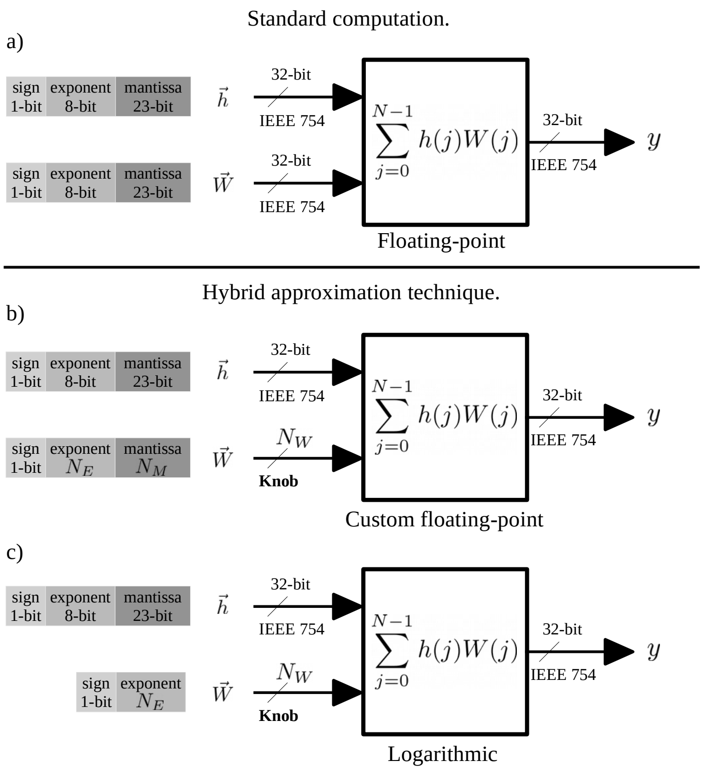


Fig. 4. Dot-product hardware module with (a) standard floating-point (IEEE 754) arithmetic, (b) hybrid custom floating-point, and (c) hybrid logarithmic approximation.

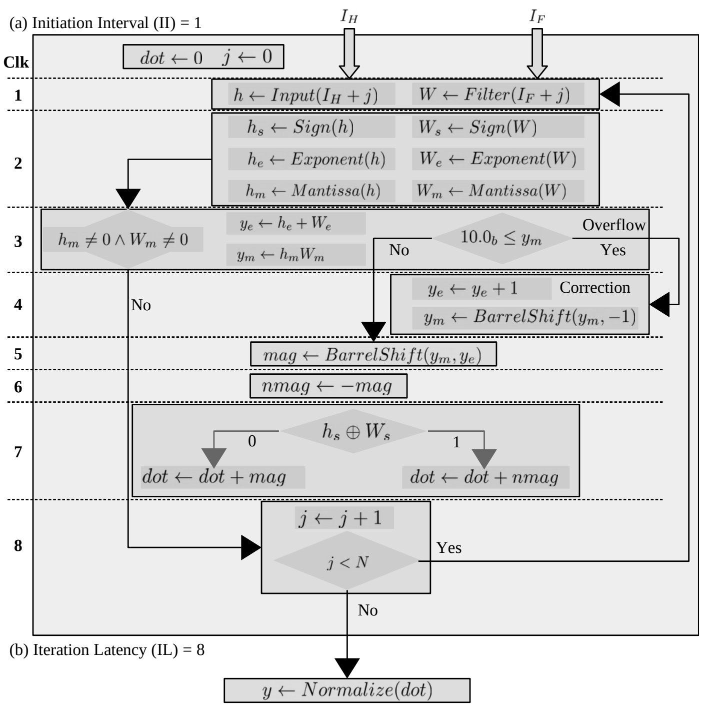
floating-point computation adopting the dot-product with hybrid custom floating-point and logarithmic approximation [9]. The hardware dot-product is illustrated in **Fig.** 4. This approach: (1) denormalizes input values, (2) executes computation with integer format for exponent and mantissa, and finally, (3) it normalizes the result into IEEE 754 format, see **Fig.** 5. Rather than a parallelized structure, this is a pipelined hardware design suitable for resource-limited devices. The latency in clock cycles of this hardware module is defined by **Eq.** (3) and **Eq.** (4), where N is the dot-product vector length. The latency equations are obtained from the general pipelined hardware latency formula: $L = (N-1)II + IL$, where II is the initiation interval (**Fig.** 5(a)), and IL is the iteration latency (**Fig.** 5(b)). Both II and IL are obtained from the high-level synthesis analysis. The logarithmic approximation removes the mantissa bit-field, which removes the mantissa multiplication and correction in clock cycle 3 and 4, respectively, see **Fig.** 5.

$$L_{custom} = N + 7 \tag{3}$$

$$L_{log} = N + 6 \tag{4}$$

As a design parameter, both the exponent and mantissa bit-width of the weight/filter vector provides a tunable knob to trade-off between resource utilization and QoR [18]. These parameters must be defined before hardware synthesis.

*3) **On-chip memory utilization**:* The total on-chip memory utilization on the TP is defined by **Eq.** (5), where $Input_M$


(a) Initiation Interval (II) = 1
I_H
I_F
Clk
$dot \leftarrow 0 \quad j \leftarrow 0$
1
$h \leftarrow Input(I_H + j)$
$W \leftarrow Filter(I_F + j)$
2
$h_s \leftarrow Sign(h)$
$W_s \leftarrow Sign(W)$
$h_e \leftarrow Exponent(h)$
$W_e \leftarrow Exponent(W)$
$h_m \leftarrow Mantissa(h)$
$W_m \leftarrow Mantissa(W)$
3
$h_m \neq 0 \wedge W_m \neq 0$
$y_e \leftarrow h_e + W_e$
$y_m \leftarrow h_m W_m$
$10.0_b \leq y_m$
Overflow
No
Yes
4
No
$y_e \leftarrow y_e + 1$
Correction
$y_m \leftarrow BarrelShift(y_m, -1)$
5
$mag \leftarrow BarrelShift(y_m, y_e)$
6
$nmag \leftarrow -mag$
7
$h_s \oplus W_s$
0
1
$dot \leftarrow dot + mag$
$dot \leftarrow dot + nmag$
8
$j \leftarrow j + 1$
$j < N$
Yes
No
(b) Iteration Latency (IL) = 8
$y \leftarrow Normalize(dot)$

$$y_{(I_H, I_F, N)} = \sum_{j=0}^{N-1} h_{(I_H+j)} W_{(I_F+j)}$$

Fig. 5. Dot-product hardware module with hybrid custom floating-point, (a) exhibits the initiation interval of 1 clock cycle, and (b) presents the iteration latency of 8 clock cycles. I_H and I_F represent the input and filter buffer indexes, respectively.

is the *input buffer*, $Filter_M$ is the *filter buffer*, $Bias_M$ is the *bias buffer*, and V_M represents the local variables required for operation. The on-chip memory buffers are defined in bits. **Fig.** 3 illustrates the convolution operation utilizing the on-chip memory buffers.

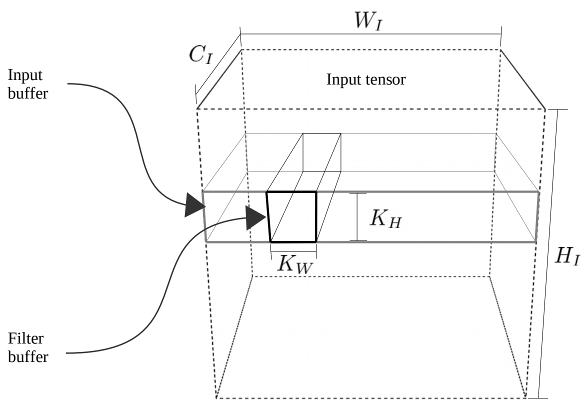


Fig. 6. Design parameters for on-chip memory buffers on the TP.

$$TP_M = Input_M + Filter_M + Bias_M + V_M \tag{5}$$

The memory utilization of *input buffer* is defined by **Eq.** (6), where K_H is the height of the convolution kernel, W_I is the width of the input tensor, C_I is the number of input channels, and $BitSize_I$ is the bit size of each input tensor element.

$$Input_M = K_H W_I C_I BitSize_I \tag{6}$$

The memory utilization of *filter buffer* is defined by **Eq.** (7), where K_W and K_H are the width and height of the convolution kernel, respectively; C_I and C_O are the number of input and output channels, respectively; and $BitSize_F$ is the bit size of each filter element.

$$Filter_M = C_I K_W K_H C_O BitSize_F \tag{7}$$

The memory utilization of *bias buffer* is defined by **Eq.** (8), where C_O is the number of output channels, and $BitSize_B$ is the bit size of each bias element.

$$Bias_M = C_O BitSize_B \tag{8}$$

As a design trade-off, **Eq.** (9) defines the capacity of output channels based on the given design parameters. The total on-chip memory TP_M determines the TP capacity.

$$C_O = \frac{TP_M - V_M - K_H W_I C_I BitSize_I}{C_I K_W K_H BitSize_F + BitSize_B} \tag{9}$$

The number formats implemented in the TP are defined by $BitSize_F$, $BitSize_B$ and $BitSize_I$. For example, a 5-bit custom floating-point format can be defined by 1-bit sign, 3-bit exponent and 1-bit mantissa. These are design parameters defined before hardware synthesis. This allows fine control of BRAM utilization, suitable for resource-limited devices.

C. Quantized aware training

The quantize-aware training method is an iterative optimization. The custom CNN model is initially trained with early stop monitoring until minimal validation loss, then the CNN model is retrained including the quantization method implemented as a callback function on every batch end, see **Algorithm** 1. The quantization method maps the full precision filter and bias values to the closest representable quantized values, see **Algorithm** 2. The quantize-aware training method starts with a wide exponent size target (e.g. 5-bits) and gradually reduces the target size until the model drops to a given accuracy degradation threshold (e.g. 1%). We have observed that the exponent bit size plays a more predominant influence on the model accuracy than the mantissa bit size. The mantissa bit size can be set to the minimum (e.g. 1-bit). This method quantizes the filter and bias tensors of the *Conv2D* and *SeparableConv2D* layers. This method is integrated in TensorFlow/Keras framework. The resulting quantized parameters are truncated and buffered in the on-chip memory of the TP during *configuration* mode.

D. Embedded software architecture

The software architecture is a layered object-oriented application framework written in C++, see **Fig.** 7. The main characteristics o the software layers are as follows:

- *Application*: As the highest level of abstraction, this layer implements the embedded application logic with the ML library.

```
Algorithm 1: Training method.
  input: $MODEL$ as the CNN.
  input: $E_{size}$ as the target exponent bit size.
  input: $M_{size}$ as the target mantissa bits size.
  input: $D_{train}$ as the training data set.
  input: $D_{val}$ as the validation data set.
  input: $Acc_d$ as the accuracy degradation threshold.
  input: $Loop_{max}$ as the max quantization loop iterations.
  output: $MODEL$ as the quantized CNN.
    // Regular training with early stop
    $Train(MODEL, D_{train}, D_{val})$
    // Get benchmark accuracy
    $acc_i \leftarrow Evaluate(MODEL, D_{val})$
    // Initialize quantize training
    $acc_q \leftarrow 0, loop_c \leftarrow 0$
    while $(acc_q < acc_i - Acc_d) \wedge (loop_c < Loop_{max})$ do
      // Iterative optimization
      $callback \leftarrow Quantize(E_{size}, M_{size})$
      // Quantized-aware training with early stop
      $Train(MODEL, D_{train}, D_{val}, callback)$
      $acc_q \leftarrow Evaluate(MODEL, D_{val})$
      $loop_c \leftarrow loop_c + 1$
    end while
```

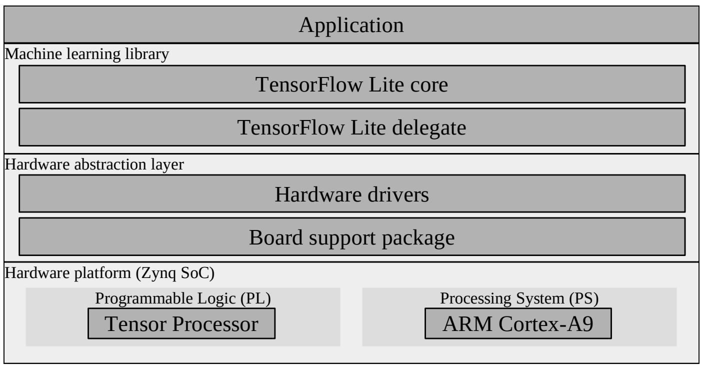


Fig. 7. Base embedded software architecture.

- *Machine learning library*: This layer consist of TensorFlow Lite micro. This offers a comprehensive high level API that allows ML inference. This provides delegate interfaces for custom hardware accelerators.
- *Hardware abstraction layer*: This layer consist of the hardware drivers to handle initialization and runtime operation of the TP and DMA.

V. Experimental results

The proposed hardware/software co-design framework is demonstrated on a Xilinx Zynq-7020 SoC (Zybo-Z7 development board). On the PL, we implement the proposed hardware architecture with a clock frequency at $150MHz$. On the PS, we execute the bare-metal software TF Lite Micro on the ARM Cortex-A9 at $666MHz$ equipped with NEON floating-point unit (FPU) [20].

```
Algorithm 2: Custom floating-point quantization method.
  input: $MODEL$ as the CNN.
  input: $E_{size}$ as the target exponent bit size.
  input: $M_{size}$ as the target mantissa bits size.
  input: $STDM_{size}$ as the IEEE 754 mantissa bit size.
  output: $MODEL$ as the quantized CNN.
    for $layer$ in $MODEL$ do
      if $layer$ is $Conv2D$ or $SeparableConv2D$ then
        $filter \leftarrow Filter(layer)$ // Get filter tensor
        $bias \leftarrow Bias(layer)$ // Get bias tensor
        for $x$ in $filter$ and $bias$ do
          $sign \leftarrow Sign(x)$
          $exp \leftarrow Exponent(x)$
          // Get full range exponent value with $E_{size}$
          $fullexp \leftarrow 2^{E_{size}-1} - 1$
          // Get custom truncated mantissa value with $M_{size}$
          $cman \leftarrow CustomMantissa(x, M_{size})$
          // Get leftover mantissa value
          $leftman \leftarrow LeftoverMantissa(x, M_{size})$
          if $exp < -fullexp$ then
            // Set minimum quantized value
            $x \leftarrow 0$
          else if $exp > fullexp$ then
            // Set maximum quantized value
            $x \leftarrow (-1)^{sign} \cdot 2^{fullexp} \cdot (1 + (1 - 2^{-Msize}))$
          else
            if $2^{STDM_{size}-M_{size}-1} - 1 < leftman$ then
              // Leftover mantissa above halfway threshold
              $cman \leftarrow cman + 1$
              if $2^{M_{size}} - 1 < cman$ then
                // Mantissa overflow
                $cman \leftarrow 0$
                $exp \leftarrow exp + 1$
              end if
            end if
            // Build custom quantized floating-point value
            $x \leftarrow (-1)^{sign} \cdot 2^{exp} \cdot (1 + cman \cdot 2^{-M_{size}})$
          end if
        end for
        $SetFiler(layer, filter)$
        $SetBias(layer, bias)$
      end if
    end for
```

To demonstrate compliance of the proposed design, we build models A and B in TensorFlow. Model B incorporates depthwise separable convolution operations (a depthwise convolution followed by a pointwise convolution). See **Fig.** 8.

To demonstrate hardware feasibility, A and B are evaluated by addressing a design exploration with the following implementations: (1) fixed-point, (2) floating-point LogiCORE, (3) hybrid custom floating-point approximation, and (4) hybrid logarithmic approximation.

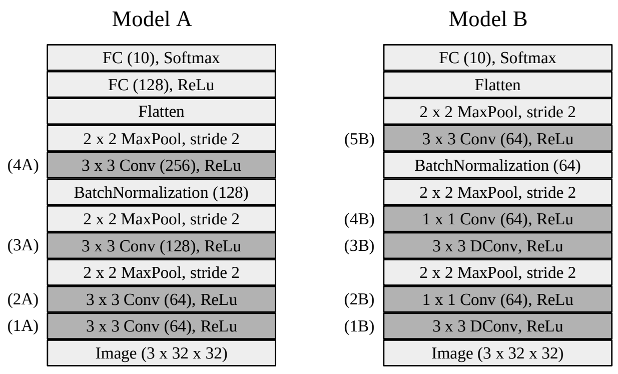


Fig. 8. CNN-based models for case study.

A. Hardware design exploration

1) **Fixed-point**: To evaluate the compute performance on fixed-point, we convert A and B to TF Lite models with 8-bit fixed-point quantization. The compute performance is presented in **Tab.** I. A runtime execution of A is illustrated in **Fig.** 9. This implementation achieves a peak runtime acceleration of $45.23\times$ in model A at the tensor operation *(4A) Conv*, see **Tab.** I.
2) **Floating-point LogiCORE**: To evaluate the compute performance on floating-point models, we convert A and B to TF Lite without quantization. The compute performance is presented in **Tab.** II. This implementation achieves a peak acceleration of $9.77\times$ in model A at the tensor operation *(4A) Conv*.
3) **Hybrid custom floating-point approximation**: This implementation presents a peak acceleration of $44.87\times$ in model A at the tensor operation *(4A) Conv*. See **Tab.** III. This implementation achieves a $4.59\times$ acceleration over the LogiCORE floating-point implementation. The runtime execution of model B with *DConv* tensor operations is illustrated in **Fig.** 11.
4) **Hybrid logarithmic approximation**: This implementation is presented for comparison in **Fig.** 10, which shows the runtime executions of model A with the proposed floating-point solutions including hybrid logarithmic approximation.

TABLE I
COMPUTE PERFORMANCE WITH FIXED-POINT ON MODEL A AND B.

Tensor operation		CPU	TP (fixed-point)			Accel.
Operation	**MMAC**	t **(ms)**	t **(ms)**	**MMAC/s**	**GMAC/W**	
Model A						
(1A) Conv	1.769	700.22	55.19	32.06	0.23	**12.69**
(2A) Conv	37.748	12,666.91	297.08	127.06	0.93	**42.64**
(3A) Conv	18.874	6,081.01	142.99	131.99	0.97	**42.53**
(4A) Conv	18.874	5,543.77	122.58	153.97	1.13	**45.23**
Model B						
(1B) DConv	0.027	13.43	0.63	43.74	0.25	**21.25**
(2B) Conv	0.196	129.95	11.57	16.98	0.12	**11.23**
(3B) DConv	0.147	69.18	3.33	44.26	0.25	**20.77**
(4B) Conv	1.048	378.78	9.96	105.25	0.77	**38.02**
(5B) Conv	2.359	694.60	16.46	143.22	1.05	**42.20**

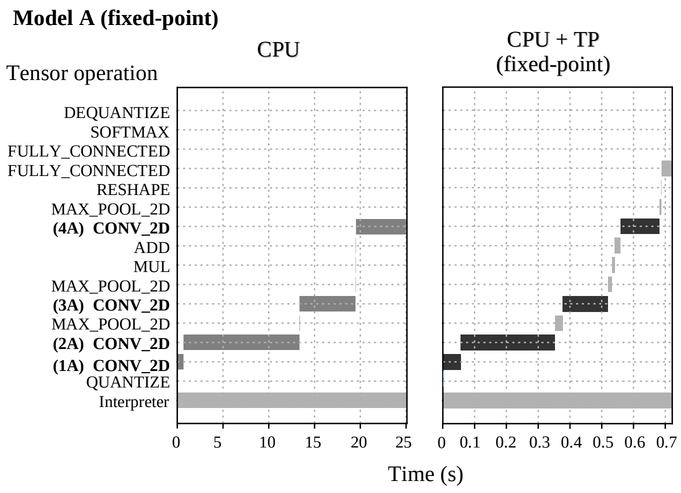


Fig. 9. Compute performance with fixed-point on model A.

TABLE II
COMPUTE PERFORMANCE WITH FLOATING-POINT LOGICORE ON MODELS A AND B.

Tensor operation		CPU	TP (floating-point LogiCORE)			Accel.
Operation	**MMAC**	t **(ms)**	t **(ms)**	**MMAC/s**	**GMAC/W**	
Model A						
(1A) Conv	1.769	670.95	120.07	14.73	0.21	**5.59**
(2A) Conv	37.748	12,722.13	1,328.08	28.42	0.40	**9.58**
(3A) Conv	18.874	6,094.85	636.53	29.65	0.42	**9.58**
(4A) Conv	18.874	5,564.79	569.30	33.15	0.47	**9.77**
Model B						
(1B) DConv	0.027	11.51	1.557	17.75	0.23	**7.39**
(2B) Conv	0.196	94.82	20.487	9.59	0.13	**4.62**
(3B) DConv	0.147	58.84	8.355	17.64	0.23	**7.04**
(4B) Conv	1.048	368.66	40.271	26.03	0.37	**9.15**
(5B) Conv	2.359	697.08	72.981	32.32	0.46	**9.55**

TABLE III
COMPUTE PERFORMANCE WITH HYBRID CUSTOM FLOATING-POINT APPROXIMATION ON MODELS A AND B.

Tensor operation		CPU	TP (H. custom floating-point)			Accel.
Operation	**MMAC**	t **(ms)**	t **(ms)**	**MMAC/s**	**GMAC/W**	
Model A						
(1A) Conv	1.769	670.95	68.50	25.83	0.39	**9.8**
(2A) Conv	37.748	12,722.13	307.83	122.63	1.85	**41.33**
(3A) Conv	18.874	6,094.85	147.97	127.55	1.93	**41.19**
(4A) Conv	18.874	5,564.79	124.03	152.17	2.30	**44.87**
Model B						
(1B) DConv	0.027	11.51	1.41	19.63	0.27	**8.17**
(2B) Conv	0.196	94.82	20.34	9.43	0.14	**4.66**
(3B) DConv	0.147	58.84	6.58	22.41	0.31	**8.94**
(4B) Conv	1.048	368.66	12.75	82.23	1.24	**28.91**
(5B) Conv	2.359	697.08	17.14	137.68	2.08	**40.68**

B. Classification accuracy

We evaluate the classification accuracy of the CNN models under the effects of custom floating and logarithmic quantization. **Tab.** IV presents the list of custom formats proposed for evaluation. In this case, the *filter* and *bias* tensors are quantized from base floating-point representation (IEEE 754) into custom reduced formats with bit-truncation and -rounding methods. For this evaluation, we train A an B for image classification

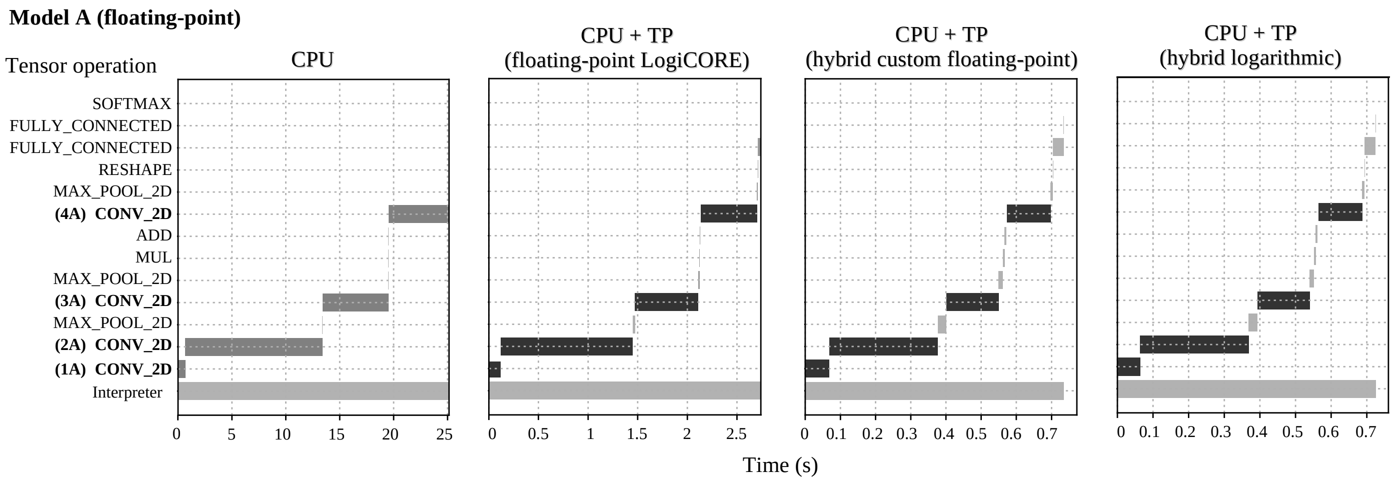


Fig. 10. Compute performance with the proposed floating-point solutions on model A.

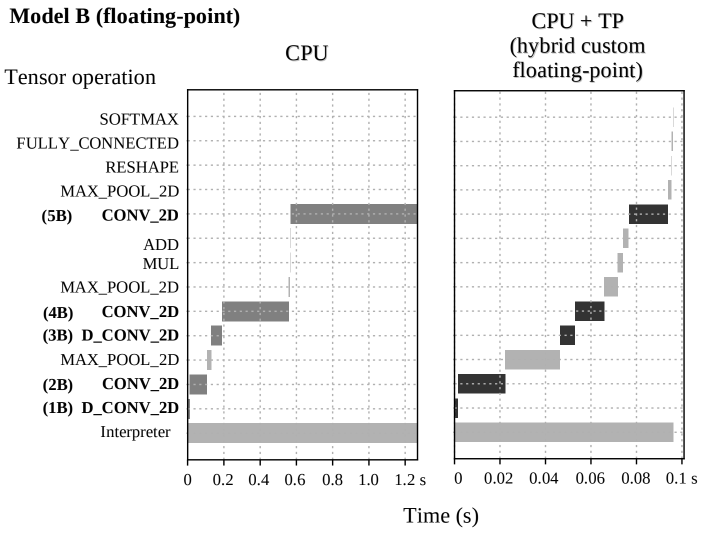


Fig. 11. Compute performance on model B (floating-point).

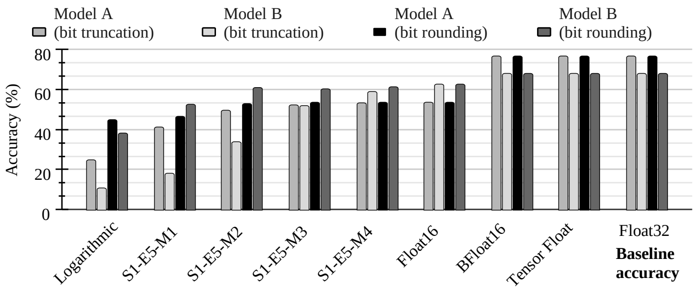


Fig. 12. Accuracy performance using hybrid custom floating-point approximation with various formats. Samples: CIFAR-10 test dataset (10, 000 images).

with CIFAR-10 dataset. We deploy the models with a baseline accuracy of 76.6% for A, and 68.8% for B. See **Fig.** 12.

TABLE IV
IMPLEMENTED FLOATING-POINT FORMATS FOR ACCURACY EVALUATION.

Floating-point formats				
Name	Size (bits)	Sign	Exponent	Mantissa
Logarithmic	6	1	5	0
S1-E5-M1	7	1	5	1
S1-E5-M2	8	1	5	2
S1-E5-M3	9	1	5	3
S1-E5-M4	10	1	5	4
Float16	16	1	5	10
BFloat16	16	1	8	7
Tensor Float	19	1	8	10
Float32	32	1	8	23

C. *Resource utilization and power dissipation*

The resource utilization and power dissipation of the TP is listed in **Tab.** V. The power dissipation of the Zynq device is presented in **Fig.** 13.

TABLE V
RESOURCE UTILIZATION AND POWER DISSIPATION OF THE PROPOSED TP ENGINES.

	Post-implementation resource utilization				Power (W)
TP engine	LUT	FF	DSP	BRAM 18K	
1) Fixed-point					
Conv	5,677	4,238	78	70	0.136
DConv	7,232	5,565	106	70	0.171
Conv + DConv	12,684	8,015	160	70	0.248
2) Floating-point LogiCore					
Conv	4,670	3,909	59	266	0.070
DConv	6,263	5,264	82	266	0.075
Conv + DConv	10,871	7,726	123	266	0.119
3) Hybrid custom floating-point approximation					
Conv	6,787	4,349	56	74	0.066
DConv	8,209	5,592	79	74	0.072
Conv + DConv	14,590	8,494	117	74	0.108
4) Hybrid logarithmic approximation					
Conv	6,662	4,242	54	58	0.060
DConv	8,110	5,380	77	58	0.066
Conv + DConv	14,370	8,175	113	58	0.105

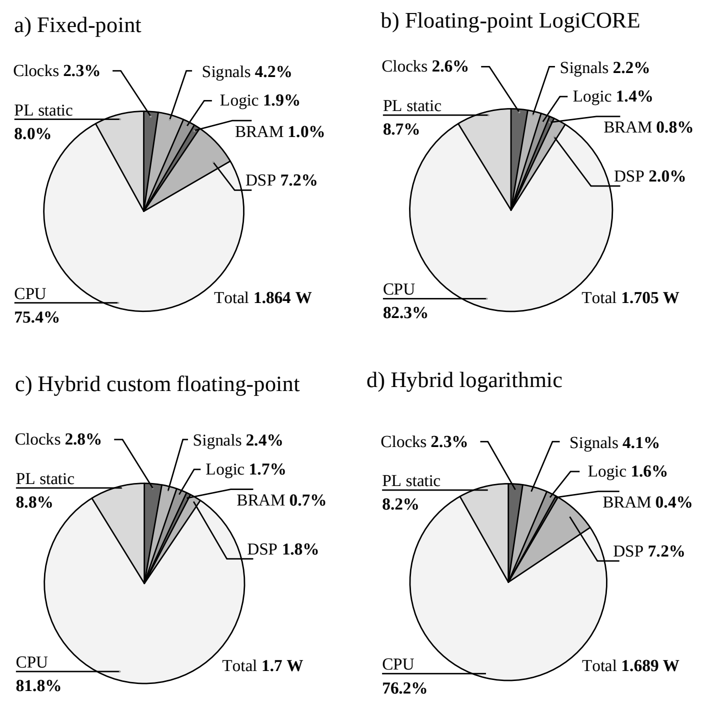


Fig. 13. Estimated power dissipation of the Zynq-7020 SoC with different TP engines.

D. Discussion

1) **Energy consumption**: The implementations with hybrid custom floating-point and logarithmic approximation are the most efficient with energy reduction of $954\times$ and $1,055\times$, respectively. **Tab.** VI presents the energy-delay product (EDP) and energy reduction in *(4A) Conv* operator.

TABLE VI
ENERGY CONSUMPTION IN TENSOR OPERATION *(4A) Conv*.

Engine	t **(ms)**	**Power (W)**	**EDP (J)**	**Reduction**
CPU	5,564.79	1.404	7,812.97	1.00
Fixed-point	122.58	0.136	16.67	468.66
Floating-point LogiCORE	569.30	0.070	39.85	196.05
Hybrid custom floating-point	124.03	0.066	8.19	**954.43**
Hybrid logarithmic	123.32	0.060	7.40	**1,055.92**

2) **Resource utilization**: The fixed-point implementation presents the highest DSP utilization. Hence, this TP presents the highest power dissipation.
3) **Accuracy**: The hybrid custom floating-point approximation presents the best trade off between QoR and energy-efficiency. The bfloat16 (brain floating-point with 16-bits) achieves a comparable QoR with floating-point 32-bits, see **Fig.** 12. To improve accuracy, the CNN models would require quantization aware training methods.
4) **Bottleneck**: To increase performance, this implementation would require matching computational throughput with memory bandwidth using parallelization approaches.

VI. Conclusions

In this paper, we present a tensor processor as a dedicated hardware accelerator for TensorFlow Lite on embedded FPGA. We accelerate *Conv2D* and *DepthwiseConv2D* tensor operations for fixed-point and floating-point computation. The proposed optimization technique performs vector dot-product with hybrid custom floating-point and logarithmic approximation. This approach accelerates computation, reduces energy consumption and resource utilization. To demonstrate the potential of the proposed architecture, we presented a design exploration with four compute engines: (1) fixed-point, (2) Xilinx floating-point LogiCORE IP, (3) hybrid custom floating-point approximation, and (4) hybrid logarithmic approximation.

A single tensor processor running at 150 MHz on a Xilinx Zynq-7020 achieves $45\times$ runtime acceleration and $954\times$ power reduction on $Conv2D$ tensor operation compared with ARM Cortex-A9 at 666MHz, and $4.59\times$ compared with the equivalent implementation with floating-point LogiCORE IP.

References

[1] G. Li, C. Deng, J. Wu, X. Xu, X. Shao, and Y. Wang, "Sensor data-driven bearing fault diagnosis based on deep convolutional neural networks and s-transform," *Sensors*, vol. 19, no. 12, p. 2750, 2019.
[2] F. Dong, X. Yu, E. Ding, S. Wu, C. Fan, and Y. Huang, "Rolling bearing fault diagnosis using modified neighborhood preserving embedding and maximal overlap discrete wavelet packet transform with sensitive features selection," *Shock and Vibration*, vol. 2018, 2018.
[3] T. Nagayama and B. F. Spencer Jr, "Structural health monitoring using smart sensors," Newmark Structural Engineering Laboratory. University of Illinois at Urbana , Tech. Rep., 2007.
[4] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2019.
[5] Y. C. Kim, H.-G. Yu, J.-H. Lee, D.-J. Park, and H.-W. Nam, "Hazardous gas detection for ftir-based hyperspectral imaging system using dnn and cnn," in *Electro-Optical and Infrared Systems: Technology and Applications XIV*, vol. 10433. International Society for Optics and Photonics, 2017, p. 1043317.
[6] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 5–14.
[7] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.
[8] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.
[9] Y. Nevarez, D. Rotermund, K. R. Pawelzik, and A. Garcia-Ortiz, "Accelerating spike-by-spike neural networks on fpga with hybrid custom floating-point and logarithmic dot-product approximation," *IEEE Access*, 2021.
[10] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.
[11] A. Yazdanbakhsh, K. Seshadri, B. Akin, J. Laudon, and R. Narayanaswami, "An evaluation of edge tpu accelerators for convolutional neural networks," *arXiv preprint arXiv:2102.10423*, 2021.
[12] "Coral. dev board datasheet.," https://coral.ai/docs/dev-board/datasheet/., accessed September 15, 2021.
[13] S. Cass, "Taking ai to the edge: Google's tpu now comes in a maker-friendly package," *IEEE Spectrum*, vol. 56, no. 5, pp. 16–17, 2019.
[14] P. Xilinx, "Zynq dpu v3.1, product guide," 2019.
[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] "Tensorflow lite for microcontrollers," https://github.com/tensorflow/tflite-micro.

[17] J. Hrica, "Floating-point design with vivado hls," *Xilinx Application Note*, 2012.

[18] J. Park, J. H. Choi, and K. Roy, "Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 5, pp. 787–793, 2009.

[19] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[20] U. Xilinx, "Zynq-7000 all programmable soc: Technical reference manual," 2015.