Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

SensorEdge: Design Exploration Framework for Near-Sensor Analytics on Resource-Constrained Embedded FPGAs

Corresponding author: Yarib Nevarez (e-mail: nevarez@item.uni-bremen.de).

This work is funded by the Consejo Nacional de Ciencia y Tecnologia - CONACYT

ABSTRACT The use of artificial intelligence (AI) in near-sensor analytic applications is entering a new era based on the use of ubiquitous small connected devices. This transformation requires the adoption of design techniques that reconcile accurate results with sustainable system architectures. In this paper, we present a design methodology for training and deployment of machine learning (ML) models with scalable hardware acceleration targeting low-power and resource-limited embedded FPGAs. The key contributions of this work are the implementation of custom reduced floating-point quantization and its dedicated hardware design for low-power sensor analytics applications. We propose a quantization aware training method that improves the generalization of convolutional neural networks (CNNs) increasing overall accuracy using less than 8-bits floating-point quantization on trainable parameters. As hardware accelerator, we propose a fully customizable tensor processor (TP) implementing a pipelined vector dot-product with hybrid custom floating-point and logarithmic approximation. This approach reduces energy consumption and resource utilization preserving inference accuracy. We demonstrate our methodology implementing a CNN-based sensor analytic application for structural health monitoring (SHM) for anomaly localization. The embedded hardware/software framework is demonstrated on XC7Z007S. The TP achieves a peak power efficiency of 4.5 GFLOP/s/W, runtime acceleration of 55X on Conv2D tensor operators without accuracy degradation.

INDEX TERMS 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 edge computing and 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.

In recent years, there is an increasing demand to introduce on-chip AI-based analytics into the smart city and Industry 4.0 infrastructure [6]. The paradigm of edge computing provides new solutions by bringing intelligence closer to the data source. This approach preserves sensitive and private data on devices, and provides low latency, energy efficiency, and scalability compared to cloud services while reducing the network bandwidth [7]. Moreover, these solutions are boosted by advances in ML models, processing power, and big data.

CNN-based models, as one of the main types of artificial neural networks (ANNs), have been successfully used in sensor analytics with automatic feature learning from sensory data [8]–[11]. In this context, CNN models are applied for automatic feature learning, mostly, from 1D time series signals as well as for 2D time-frequency spectrograms. CNN models provide advantages over other methods, such as local dependency and scale invariance. However, these models represent computationally-intensive and power-hungry tasks, particularly, for embedded system architectures.

Due to the high computational demands of CNNs, dedicated hardware architectures are typically required to accelerate execution and improve power efficiency. 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) [12]. 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.

However, most FPGA accelerators have been implemented to target mid- to high-range FPGAs for computationally 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 unsustainable or even impossible for near-sensor analytics on low-power resource-constrained edge devices.

In the same line, model quantization is an optimization technique normally used for inference in embedded systems. This approach reduces memory footprint and hardware resource utilization, with a reasonable accuracy degradation. Quantization is achieved by lowering the precision and reducing the bit-with of the numerical representation of trainable parameters. 8-bit quantized parameters perform inference efficiently with an acceptable loss of precision [4]. Some quantization techniques can replace regular multiplication operations with simple addition and shift operations or even the total elimination of hardware multipliers for binary and ternary weights. These implementations can further reduce space in the logic circuit and improve the parallelization of logic operations [13]. However, aggressive quantization techniques can cause undesirable significant accuracy degradation.

In this article, we present a design exploration framework for floating-point shallow CNN acceleration targeting low-power, resource-limited FPGAs. The embedded software integrates TensorFlow (TF) Lite library with delegate interface to accelerate *Conv2D* tensor operations. We propose a customizable tensor processor (TP) with fully parametrized on-chip memory utilization suitable for small footprint FPGAs. To accelerate floating-point computation, we employ the pipelined hardware vector dot-product with hybrid custom floating-point and logarithmic approximation technique [14]. Further on, we propose a quantize aware training method to preserve inference accuracy with low-precision floating-point formats.

To operate the proposed system, the user trains a custom CNN model using TensorFlow or Keras, then this model is converted into a TensorFlow Lite, finally, the model is stored in a micro SD card memory and inserted in the system slot, 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 CNNs. This is a scalable and fully 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 design computes *Conv2D* and *DepthwiseConv2D* tensor operations employing a pipelined vector dot-product using hybrid custom floating-point and 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.

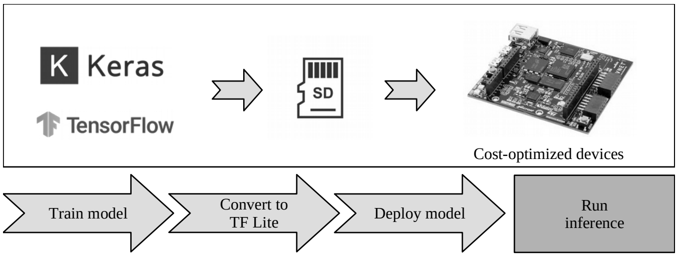

FIGURE 1. The workflow of our approach on embedded FPGAs.

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.

This design exploration framework is available to the community as an open-source project at (*hidden for double blinded review*).

II. RELATED WORK

In the literature we find plenty of hardware architectures dedicated to CNN accelerators implemented in FPGA and ASIC designs. However the related work on low-power and resource-limited devices is reduced. To the best of our knowledge, two research papers have been reported hardware implementations targeting XC7Z007S as the smallest device from Zynq-7000 SoC Family.

In [15], Chang Gao et al., presented EdgeDRNN, a recurrent neural network (RNN) accelerator for edge inference. This implementation adopts the spiking neural network (SNN) inspired delta network algorithm to exploit temporal sparsity in RNNs. However, this hardware architecture is dedicated to RNNs.

In [16], Paolo Meloni et al., presented a CNN inference accelerator for compact and cost-optimized devices. This implementation uses fixed-point for processing light-weight

CNN architectures with a power efficiency between 2.49 to 2.98 GOPS/s/W.

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 [17]. We denote *Conv* as *Conv2D* operator.

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

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.

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

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.

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

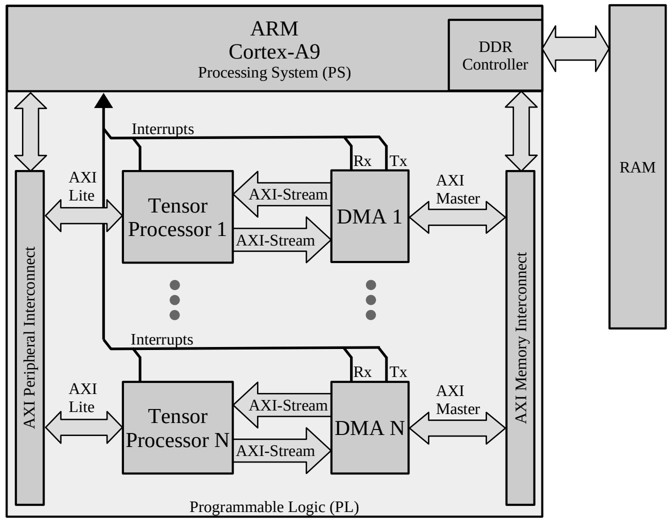

FIGURE 2. Base embedded system architecture.

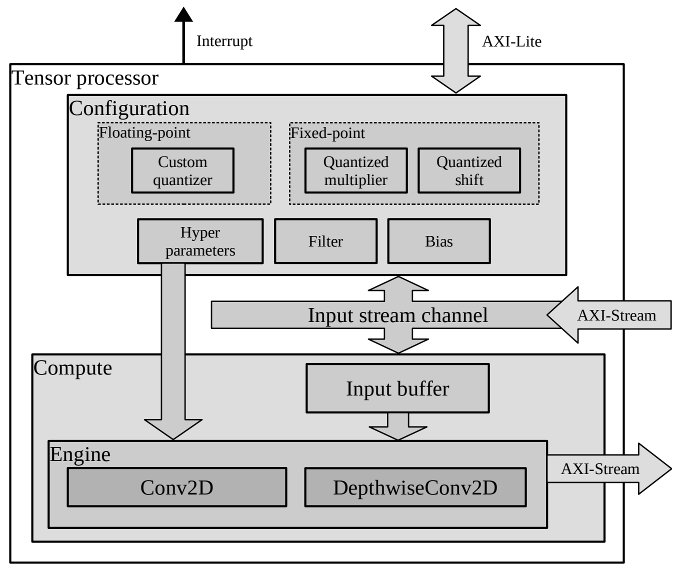

FIGURE 3. Hardware architecture of the proposed tensor processor.

This hardware architecture is implemented with high-level synthesis (HLS). The tensor operations are implemented based on the C++ TensorFlow Lite micro kernels.

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 floating-point computation adopting the dot-product with hybrid custom floating-point and logarithmic approximation [14]. 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 $Input_M$ 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.

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

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.

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

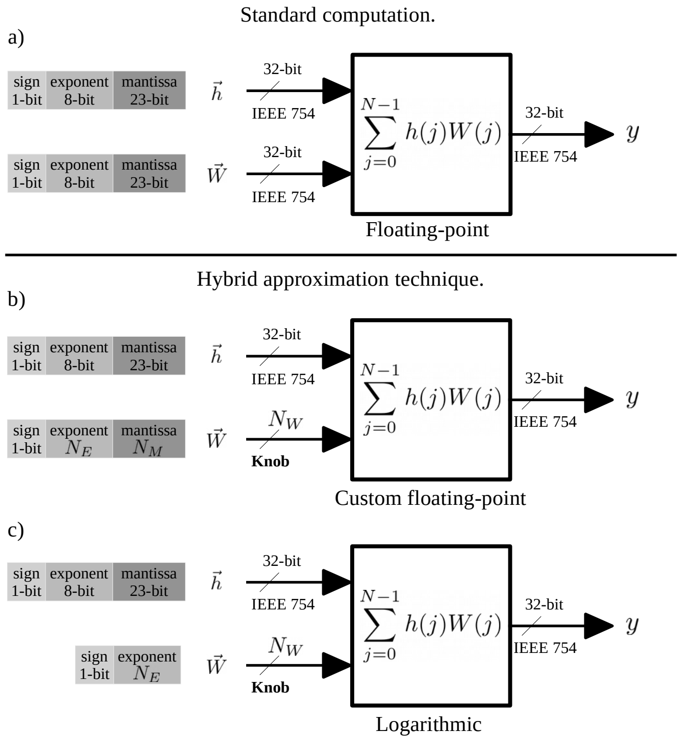

FIGURE 4. Dot-product hardware module with (a) standard floating-point (IEEE 754) arithmetic, (b) hybrid custom floating-point, and (c) hybrid logarithmic approximation.

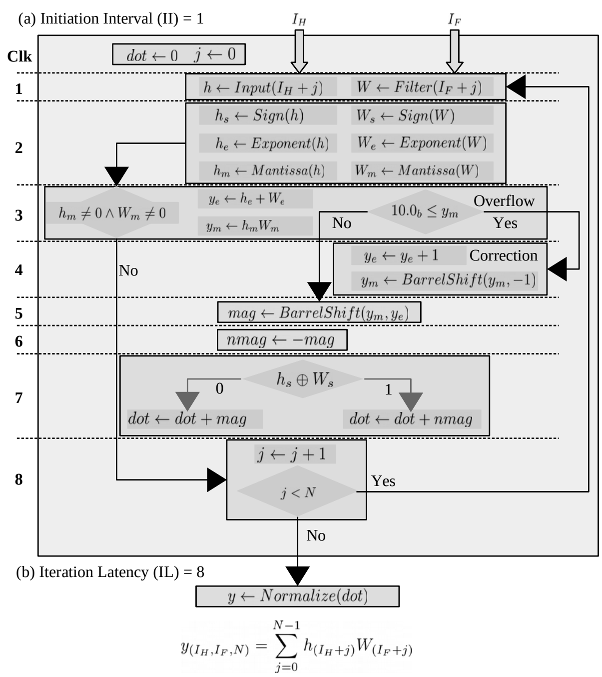

FIGURE 5. Pipelined hardware module for vector dot-product with hybrid custom floating-point, (a) exhibits the initiation interval of 1 clock cycle, and (b) presents the iteration latency of 8 clock cycles. I_H and I_F represent the input and filter buffer indexes, respectively.

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

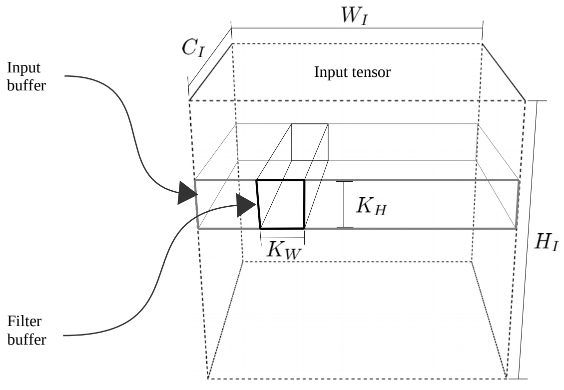

FIGURE 6. Design parameters for on-chip memory buffers on the TP.

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:

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.
$Train(MODEL, D_{train}, D_{val})$ // Regular training
$acc_i \leftarrow Evaluate(MODEL, D_{val})$ // Benchmark
$acc_q \leftarrow 0, loop_c \leftarrow 0$ // Initialize quantize training
while $(acc_q < acc_i - Acc_d) \wedge (loop_c < Loop_{max})$ **do**
// Iterative optimization
$callback \leftarrow Quantize(E_{size}, M_{size})$
$Train(MODEL, D_{train}, D_{val}, callback)$
$acc_q \leftarrow Evaluate(MODEL, D_{val})$
$loop_c \leftarrow loop_c + 1$
end while

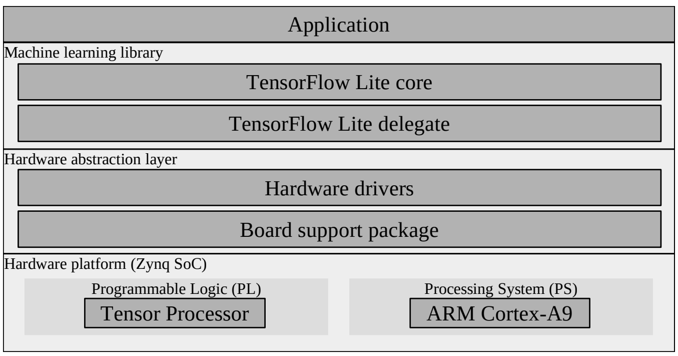

FIGURE 7. Base embedded software architecture.

- *Application*: As the highest level of abstraction, this layer implements the embedded application logic with the ML library.
- *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 XC7Z007S with 1 TP instance, and XC7Z010 with 2 TP instances. On the PL, we implement the proposed hardware architecture with a clock frequency at $200MHz$. On the PS, we execute the bare-metal software architecture on the ARM Cortex-A9 at $666MHz$ in both devices.

To demonstrate the proposed design, we build models

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**
    $filter, bias \leftarrow GetWeights(layer)$
    for x in $filter$ and $bias$ **do**
      $sign \leftarrow GetSign(x)$
      $exp \leftarrow GetExponent(x)$
      $fullexp \leftarrow 2^{E_{size}-1} - 1$ // Get full range value
      $cman \leftarrow GetCustomMantissa(x, M_{size})$
      $leftman \leftarrow GetLeftoverMantissa(x, M_{size})$
      if $exp < -fullexp$ **then**
        $x \leftarrow 0$
      else if $exp > fullexp$ **then**
        $x \leftarrow (-1)^{sign} \cdot 2^{fullexp} \cdot (1 + (1 - 2^{-Msize}))$
      else
        if $2^{STDM_{size}-M_{size}-1} - 1 < leftman$ **then**
          $cman \leftarrow cman + 1$ // Above halfway
          if $2^{M_{size}} - 1 < cman$ **then**
            $cman \leftarrow 0$ // Correct mantissa overflow
            $exp \leftarrow exp + 1$
          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
    $SetWeights(layer, filter, bias)$
  end if
end for

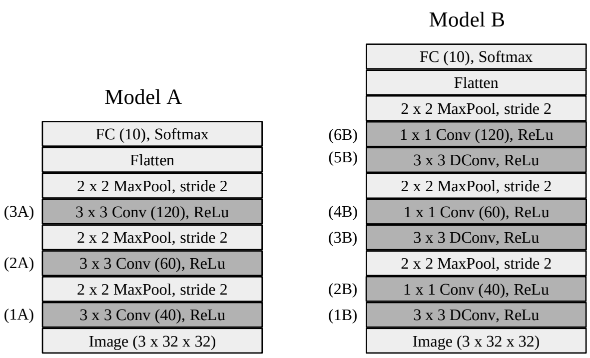

FIGURE 8. Shallow CNN models for case study.

A and B in TensorFlow. Model B incorporates depthwise separable convolution operations (a depthwise convolution followed by a pointwise convolution). See **Fig.** 8.

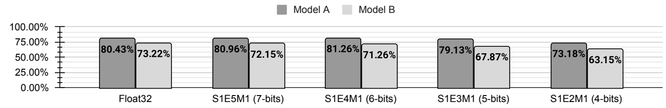

FIGURE 9. Accuracy performance using the proposed training method.

A. CUSTOM FLOATING-POINT FORMAT BASED ON CLASSIFICATION ACCURACY

To obtain the best number format, we train A and B with CIFAR-10 using early stop and batch size of 20, and *adam* optimizer. The proposed quantized-aware training method is used with two iterations, see **Fig.** 9.

To demonstrate hardware feasibility, A and B are evaluated by addressing a design exploration with hybrid custom floating-point and hybrid logarithmic approximation. We explore three reduced floating-point formats for filter and bias: exponent $E_{size} = 5, 4, 3$-bits, all formats with mantissa $M_{size} =$ 1-bit and sign $S_{size} =$ 1-bit. For Logarithmic approximation, we remove the mantissa bit.

B. HARDWARE DESIGN EXPLORATION

To evaluate the methodology, we employ **Eq.** (9), giving the maximum hyper parameters from models A and B: $W_I = 32$, $C_I = 60$, $C_O = 120$, $K_W = K_H = 3$. For the number formats, $BitSize_I = 32$-b, and $BitSize_F = BitSize_B =$ 6-bits. To determine V_M, we use HLS tool, which gives an estimate of 6 RAM blocks. The performance evaluation and the hardware resource utilization are displayed in **Tab. ??** and **Tab.** 1, respectively.

1) **XC7Z007S**: As a resource-limited FPGA, this device has a capacity of 14,400 LUTs and 1.8Mb of BRAM. This limitation allows to instantiate one TP with *Conv* due to its LUT capacity. With **Eq.** (5), we obtain a BRAM utilization of 789.84Kb. This implementation presents a peak runtime acceleration of $55\times$ in model A at the tensor operation *(3A) Conv* with a power reduction of $808\times$.
2) **XC7Z010**: This device has a capacity of 17,600 LUTs and 2.1Mb of BRAM. These resources allow to instantiate two TPs with *Conv*, and one TP with *Conv* and *DConv* engines. With **Eq.** (5), we obtain a BRAM utilization of 1,580Kb. This implementation presents a peak runtime acceleration of $105\times$ in model A at the tensor operation *(3A) Conv* with a power reduction of $1121\times$. On model B, *(6B) Conv* presents a peak acceleration of $43.8\times$. The *DConv* tensor operator yields an acceleration of $6.75\times$, which is limited since the pipelined vector dot-product performs on channel wise.

VI. CONCLUSIONS

In this paper, we present a design exploration framework for floating-point CNNs acceleration on low-power, resource-limited embedded FPGAs. This design targets inexpensive IoT and near-sensor data analytic applications. We propose

TABLE 1. Hardware resource utilization and estimated power dissipation.

Device	TP	Post-implementation resource utilization				Power (W)
		LUT	FF	DSP	BRAM 36Kb	
XC7Z007S	1	7,939 55%	8,955 31%	20 30%	25 50%	1.44
XC7Z010	2	13,542 77%	15,279 43%	36 45%	46 76%	1.880

a scalable hardware architecture with customizable tensor processors integrated with TensorFlow Lite. The implemented hardware optimization realizes a pipelined vector dot-product using hybrid custom floating-point and logarithmic approximation with fully parametrized on-chip memory utilization. This approach accelerates computation, reduces energy consumption and resource utilization. We proposed a quantized-aware training method to maintain and increase inference accuracy with custom reduced floating-point formats. Experimental results on XC7Z007S (MiniZed) and XC7Z010 (Zybo) demonstrate peak acceleration and power efficiency of 105X and 5.5 GFLOP/s/W, respectively.

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 âĂę, 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] M. Lom, O. Pribyl, and M. Svitek, "Industry 4.0 as a part of smart cities," in *2016 Smart Cities Symposium Prague (SCSP)*. IEEE, 2016, pp. 1–6.

[7] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[8] T. Ince, S. Kiranyaz, L. Eren, M. Askar, and M. Gabbouj, "Real-time motor fault detection by 1-d convolutional neural networks," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 11, pp. 7067–7075, 2016.

[9] O. Janssens, V. Slavkovikj, B. Vervisch, K. Stockman, M. Loccufier, S. Verstockt, R. Van de Walle, and S. Van Hoecke, "Convolutional neural network based fault detection for rotating machinery," *Journal of Sound and Vibration*, vol. 377, pp. 331–345, 2016.

[10] O. Abdeljaber, O. Avci, S. Kiranyaz, M. Gabbouj, and D. J. Inman, "Real-time vibration-based structural damage detection using one-dimensional convolutional neural networks," *Journal of Sound and Vibration*, vol. 388, pp. 154–170, 2017.

[11] X. Guo, L. Chen, and C. Shen, "Hierarchical adaptive deep convolution neural network and its application to bearing fault diagnosis," *Measurement*, vol. 93, pp. 490–502, 2016.

[12] 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.

[13] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[14] 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.

[15] C. Gao, A. Rios-Navarro, X. Chen, S.-C. Liu, and T. Delbruck, "Edgedrnn: Recurrent neural network accelerator for edge inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 419–432, 2020.

[16] P. Meloni, A. Garufi, G. Deriu, M. Carreras, and D. Loi, "Cnn hardware acceleration on a low-power and low-cost apsoc," in *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2019, pp. 7–12.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[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.