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

Digital Object Identifier 10.1109/ACCESS.2017.DOI

CNN Sensor Analytics with Hybrid Float6 on Low-Power 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 embedded connected devices. This transformation requires the adoption of design techniques that reconcile accurate results with sustainable system architectures. As such, maximizing computational efficiency given limited hardware and power resources is increasingly being considered, and as a result, reduced-precision inference has emerged as a viable alternative to the IEEE 754 full precision floating-point arithmetic. In this paper, we present a design methodology for training, quantization, and deployment of convolutional neural networks (CNNs) with dedicated hardware acceleration targeting low-power and resource-limited embedded FPGAs. The key contributions of this work are the demonstration of custom reduced floating-point quantization and its dedicated hardware design for low-power resource-constrained data analytics applications. We propose a quantization aware training method for fine-tuning CNN-based models. This approach improves generalization and increases overall accuracy using less than 8-bits custom floating-point quantization on trainable parameters. As dedicated hardware design, 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. The proposed embedded hardware/software architecture is unified with TensorFlow Lite. We demonstrate our framework implementing a CNN-based sensor analytic application for structural health monitoring (SHM) for anomaly localization. The embedded hardware/software framework is demonstrated on XC7Z007S as the smallest and most inexpensive Zynq SoC device. The TP achieves a peak runtime acceleration of 55X on Conv2D tensor operators, and power efficiency of 4.5 GFLOP/s/W with 55% of hardware resource utilization.

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 sig-

nals 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 and not always feasible for resource-constrained applications.

Furthermore, there are two types of research to reduce the computational cost for CNN inference [13]: the first one is deep compression including weight pruning, weight quantization, and compression storage [14], [15]; the second type of research corresponds to a more efficient data representation, also known as quantization for dedicated circuit implementation. In this group, hardware implementations with customized floating-point computation have been proposed [13], [16], [17]. However, these implementations are inadequate for embedded applications, since the target devices are PCIe architectures. While all aforementioned works have good accuracy with retraining, more aggressive data representations such as binary [18], ternary [19], and mixed precision (2-bit activations and ternary weights) [20] may suffer from great accuracy loss even with time-consuming retraining. The afforded mentioned limitations make these implementations inadequate for data analytics in embedded applications.

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 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 [21]. 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.

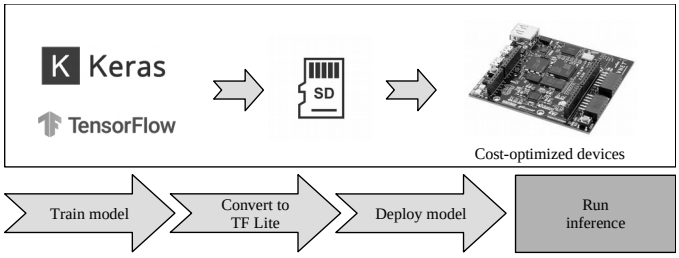

FIGURE 1. The workflow of our approach on embedded FPGAs.

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.

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

A. HARDWARE IMPLEMENTATIONS TARGETING RESOURCE-CONSTRAINED FPGAS

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 [22], 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 [23], 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.

B. HYBRID CUSTOM FLOATING-POINT QUANTIZATION

Reference [24] proposed a mixed data representation with floating-point for weights and fixed-point for activations (e.g., outputs of a layer). Reference [25] developed an 8-bit floating-point quantization scheme, which needs an extra inference batch to compensate for the quantization error. However, Reference [24] and Reference [25] did not present a circuit design for their approaches.

1) FPGA implementations

Reference [16] implements 16-bit floating-point in contrast to the 32-bit commonly used for computing. However, this implementation is inadequate for embedded applications, since the target device is a PCIe architecture. The 8-bit floating-point is also tried in FPGA [13]. Another 8-bit arithmetic, called block floating-point (BFP), is also applied [17], where a parameter has its own mantissa but shares a same exponent for one data block.

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 [26]. 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.

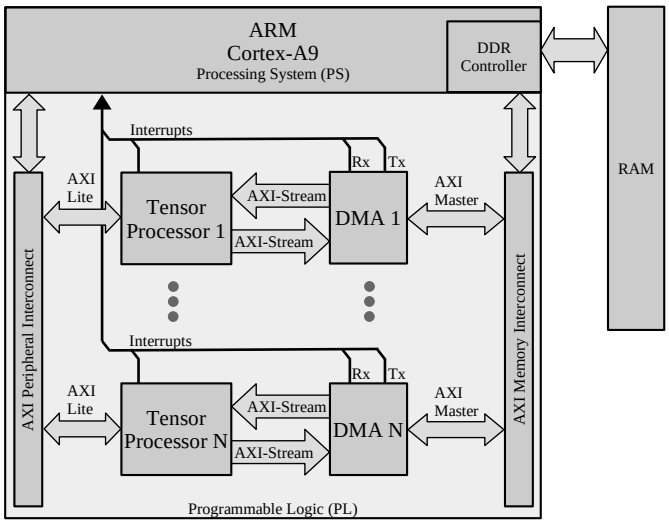

FIGURE 2. Base embedded system architecture.

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 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*.

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

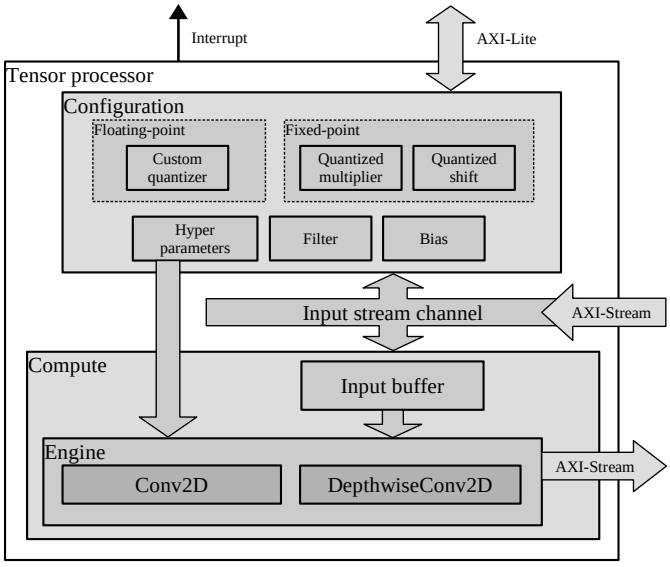

FIGURE 3. Hardware architecture of the proposed tensor processor.

- 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 [21]. 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.

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

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

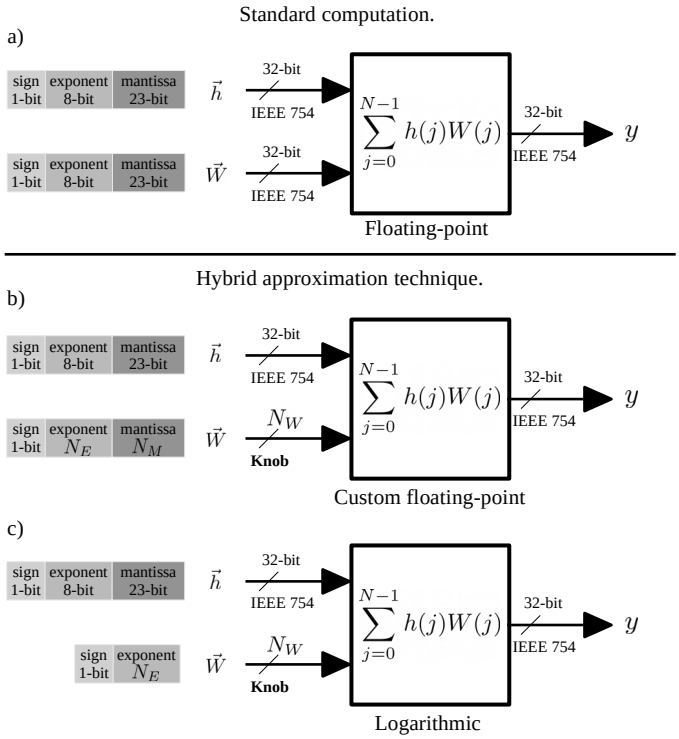

FIGURE 4. Dot-product hardware module with (a) standard floating-point (IEEE 754) arithmetic, (b) hybrid custom floating-point, and (c) hybrid logarithmic approximation.

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 [27]. 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.

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

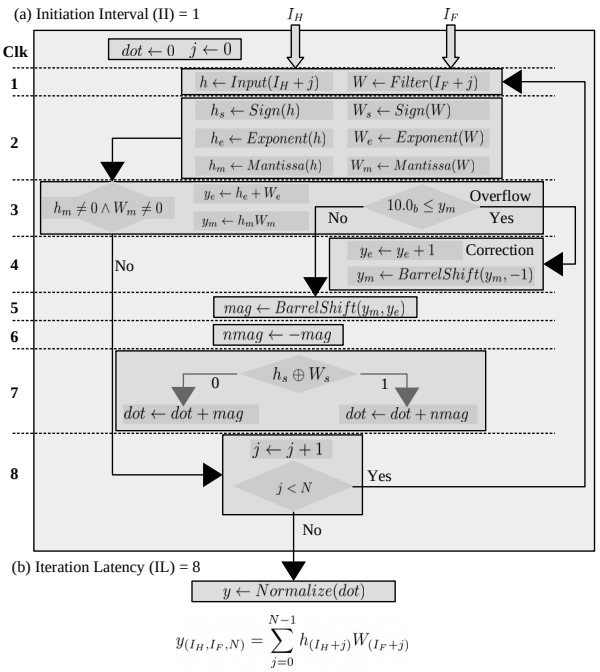

FIGURE 5. Pipelined hardware module for vector dot-product with hybrid custom floating-point, (a) exhibits the initiation interval of 1 clock cycle, and (b) presents the iteration latency of 8 clock cycles. I_H and I_F represent the input and filter buffer indexes, respectively.

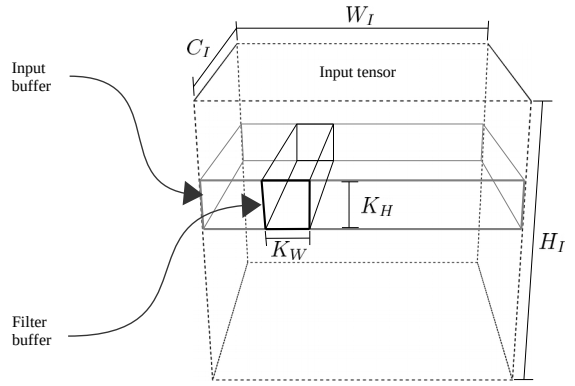

FIGURE 6. Design parameters for on-chip memory buffers on the TP.

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.

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 ← Evaluate(MODEL, D_val) // Benchmark
acc_q ← 0, loop_c ← 0 // Initialize quantize training
while (acc_q < acc_i − Acc_d) ∧ (loop_c < Loop_max) do
    // Iterative optimization
    callback ← Quantize(E_size, M_size)
    Train(MODEL, D_train, D_val, callback)
    acc_q ← Evaluate(MODEL, D_val)
    loop_c ← loop_c + 1
end while
```

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.
- *Machine learning library*: This layer consist of TensorFlow Lite micro. This offers a comprehensive high level

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 ← GetWeights(layer)
    for x in filter and bias do
      sign ← GetSign(x)
      exp ← GetExponent(x)
      fullexp ← 2^(E_size − 1) − 1 // Get full range value
      cman ← GetCustomMantissa(x, M_size)
      leftman ← GetLeftoverMantissa(x, M_size)
      if exp < −fullexp then
        x ← 0
      else if exp > fullexp then
        x ← (−1)^sign · 2^fullexp · (1 + (1 − 2^−Msize))
      else
        if 2^(STDM_size − M_size − 1) − 1 < leftman then
          cman ← cman + 1 // Above halfway
          if 2^M_size − 1 < cman then
            cman ← 0 // Correct mantissa overflow
            exp ← exp + 1
          end if
        end if
        // Build custom quantized floating-point value
        x ← (−1)^sign · 2^exp · (1 + cman · 2^−M_size)
      end if
    end for
    SetWeights(layer, filter, bias)
  end if
end for
```

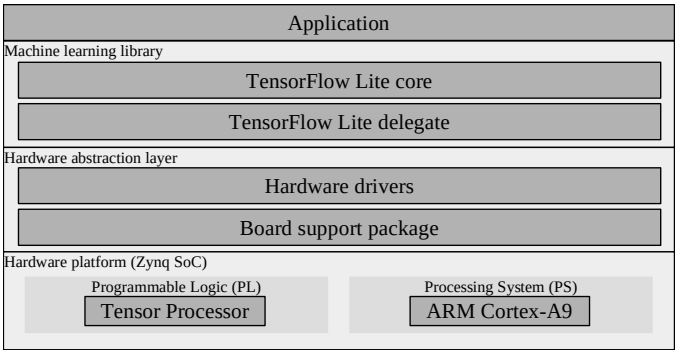

FIGURE 7. Base embedded software architecture.

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.

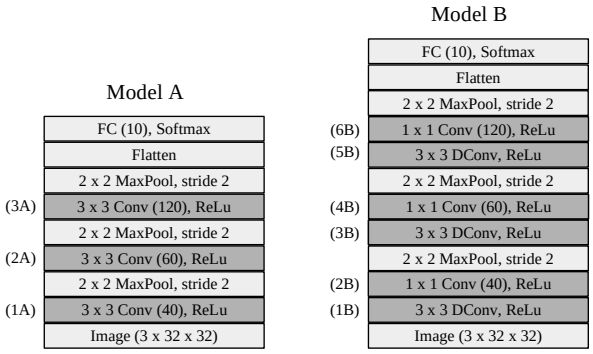

FIGURE 8. Shallow CNN models for case study.

V. EXPERIMENTAL RESULTS

The proposed hardware/software co-design framework is demonstrated on the MiniZed development board with a Zynq-7007S system-on-chip (SoC). This device integrates a single ARM Cortex-A9 processing system (PS) and programmable logic (PL) equivalent to Xilinx Artix-7 (FPGA) in a single chip [28]. The Zynq-7007S SoC architecture maps the custom logic and software in the PL and PS respectively as an embedded system.

In this platform, we implement the proposed hardware architecture to deploy the CNN model for SHM shown in **Fig.** 8. The CNN model is created, trained, and quantized using Keras/TensorFlow with Python on a desktop computer. The resulting model is converted to TensorFlow Lite, which is deployed on the MiniZed. The Zynq-7007S SoC performs the model inference with TensorFlow Lite core API running on the PS. The computational workload of the convolution layers is delegated to the TP on the PL.

For the evaluation of our approach, we address a design exploration by reviewing the computational latency, inference accuracy, resource utilization, and power dissipation. First, we benchmark the model inference on the embedded CPU, and then repeat the measurements on hardware processing units with standard floating-point computation. Afterwards, we evaluate our TP, addressing a design exploration with hybrid custom floating-point, as well as the hybrid logarithmic approximation. Finally, we present a discussion of the presented results.

A. PERFORMANCE BENCHMARK

1) Benchmark on embedded CPU

We examine the performance of the embedded CPU for inference with no hardware acceleration. In this case, the embedded software builds the CNN as a sequential model mapping the entire computation to the CPU (ARM Cortex-A9) at 666 MHz and a power dissipation of $1.658W$.

The inference on the CPU achieves a latency of $40ms$. The model is computed with standard floating-point arithmetic with no accuracy degradation. The latency and sched-

TABLE 1. Inference on embedded CPU.

Layer	Latency (ms)
HX_IN	1.184
H1_CONV	4.865
H2_POOL	3.656
H3_CONV	20.643
H4_POOL	0.828
H5_FC	3.099
HY_OUT	0.004
TOTAL	34.279

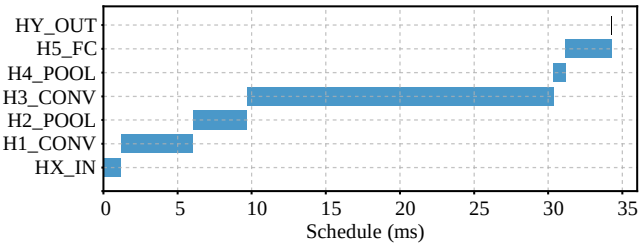

FIGURE 9. Computation on embedded CPU.

TABLE 2. Performance of TP with standard floating-point (IEEE 754) computation.

Hardware mapping		Computation schedule (ms)			
Layer	PU	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.056	0.370	0.426
H1_CONV	Conv1	0.058	0.598	2.002	2.658
H2_POOL	Pool1	0.658	0.126	1.091	1.875
	Pool2	0.785	0.125	1.075	1.985
H3_CONV	Conv2	0.911	0.280	3.183	4.374
	Conv3	1.193	0.279	3.176	4.648
H4_POOL	Pool3	1.473	0.037	0.481	1.991
H5_FC	FC	1.512	0.101	1.118	2.731
HY_OUT	CPU	1.615	0.004	0	1.619

ule of the CNN inference are displayed in **Tab.** 1 and **Fig.** 9 respectively.

2) Benchmark on tensor processor with standard floating-point computation

To benchmark the computation on hardware TP with standard floating-point, we implement the system architecture with one TP. In this case, the embedded software builds the CNN as a sequential model mapping *Conv2D* tensor operations to the TP at 200 MHz as clock frequency. The hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 2 and **Fig.** 10.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 3.

The TP instantiates an on-chip weight matrix of $52,000$ entries, wish is sufficient to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $B \in \mathbb{R}^{5\times5\times32\times64}$ for weight and bias, respectively. In order to reduce BRAM utilization, we use a custom floating-point representation composed of 4-bit exponent and 4-bit mantissa. Each 8-bit entry is promoted to its standard floating-point representation for computation.

The implementation of dot-product with standard floating-point arithmetic (IEEE 754) utilizes proprietary multiplier and adder floating-point operator cores. Vivado HLS accomplishes floating-point arithmetic operations by mapping them onto Xilinx LogiCORE IP cores, these floating-point operator cores are instantiated in the resultant RTL [29]. In this case, the implementation of the dot-product with the standard floating-point computation reuses the multiplier and adder cores already instantiated in other compute sections of the TP. The post-implementation resource utilization and power dissipation of the floating-point operator cores are shown in **Tab.** 4.

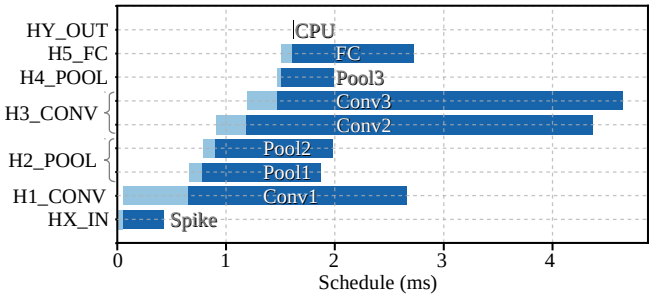

FIGURE 10. Performance of TP with standard floating-point (IEEE 754) computation.

TABLE 3. Resource utilization and power dissipation with standard floating-point (IEEE 754) computation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Spike	2,640	4,903	2	2	38
Conv	2,765	4,366	19	37	89
Pool	2,273	3,762	5	3	59
FC	2,649	4,189	8	9	66

TABLE 4. Resource utilization and power dissipation of multiplier and adder floating-point (IEEE 754) operator cores.

Core operation	DSP	FF	LUT	Latency (clk)	Power (mW)
Multiplier	3	151	325	4	7
Adder	2	324	424	8	6

B. DESIGN EXPLORATION WITH HYBRID CUSTOM FLOATING-POINT AND LOGARITHMIC APPROXIMATION

In this section, we address a design exploration to evaluate our approach for inference using hybrid custom floating-point and logarithmic approximation. First, we examine the weight matrix of each convolution layer in order to determine the minimum requirements for numeric representation and memory storage. Second, we implement the TP using the minimal floating-point and logarithmic representation as design parameters. Finally, we evaluate the overall performance, inference accuracy, resource utilization, and power dissipation.

1) Parameters for numeric representation of weight matrix

We obtain information for the numerical representation of the synaptic weight matrices from their $\log_2$-histograms

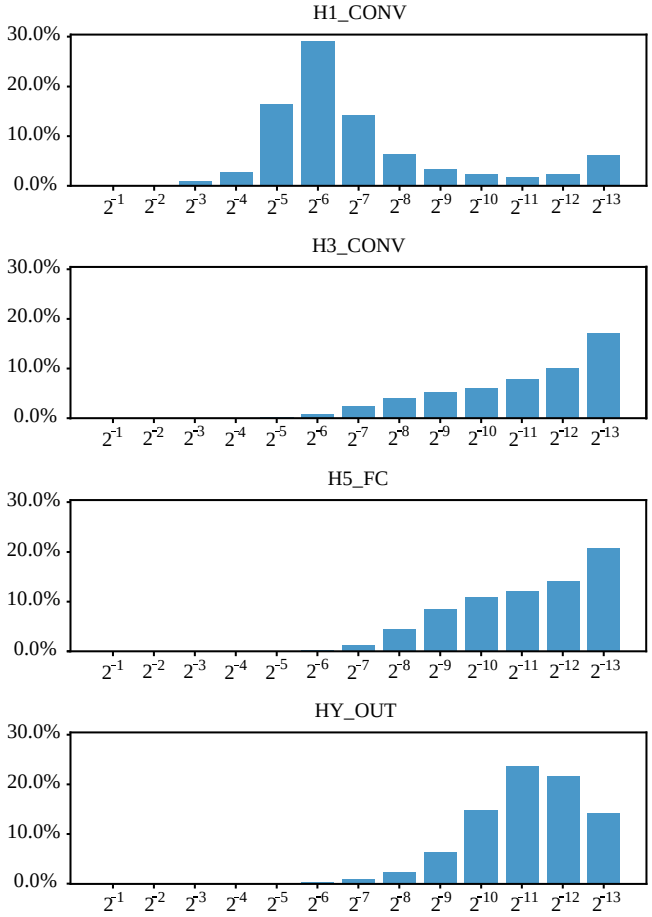

FIGURE 11. $\log_2$-histogram of each synaptic weight matrix showing the percentage of matrix elements with given integer exponent.

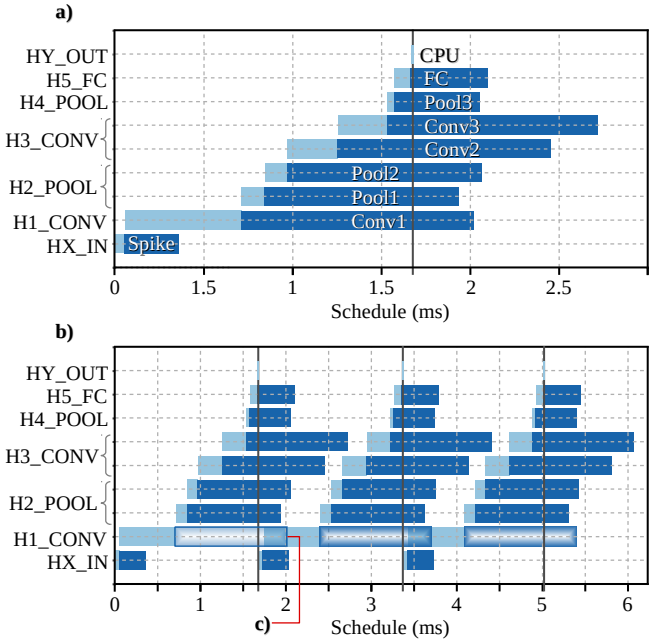

FIGURE 12. Performance on processing units with hybrid custom floating-point approximation, (a) exhibits computation schedule, (b) presents cyclic computation schedule, and (c) shows the performance of *Conv2* from a previous computation cycle during the preprocessing of *H1_CONV* on the current computation cycle without bottleneck.

presented in **Fig.** 11. These histograms show the distribution of weight values in each matrix. We observe that the minimum integer exponent value is -13. Hence, applying **Eq. (??)** and **Eq. (??)** to the given CNN, we obtain $E_{\min} = -13$ and $N_E = 4$, respectively. Therefore, 4-bits are required for the absolute binary representation of the exponents.

For quality configurability, the mantissa bit-width is a knob parameter that is tuned by the designer. This procedure leverages the builtin error-tolerance of neural networks and performs a trade-off between resource utilization and QoR. In the following subsection, we present a case study with 1-bit mantissa corresponding to the custom floating-point approximation.

2) Design exploration for dot-product with hybrid custom floating-point approximation

For this design exploration, we use a custom floating-point representation composed of 4-bit exponent and 1-bit mantissa. This format is used for both the filter matrix and bias vectors of each convolution layer. The TP instantiates on-chip stationary both the filter matrix and bias vectors for X and Y entries of 6-bit (S1E4M1). The available memory size is large enough to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $W \in \mathbb{R}^{5\times5\times32\times64}$ for $\vec{F}$ and $\vec{b}$, respectively. The hardware mapping and the computation schedule of this implementation are displayed in **Tab.** 6 and **Fig.** 12.

As shown in the computation schedule in **Tab.** 6 and **Fig.** 12, this implementation achieves a peak acceleration of 55x, and a power efficiency of 5.5 GFLOP/s/W. This configuration achieves an accuracy of 90.97% correct regressions on the 500 validation samples. This indicates an accuracy gain of 0.33%.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 5.

TABLE 5. Resource utilization and power dissipation of processing units with hybrid custom floating-point approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,139	4,850	19	25	82
FC	3,265	5,188	8	9	66

3) Design exploration for dot-product whit hybrid logarithmic approximation

As the most efficient setup and yet the worst-case quality configuration, we use a 4-bit integer exponent for logarithmic representation of $\vec{F}$ and $\vec{b}$. The hardware mapping and the computation schedule of this implementation are displayed

TABLE 6. Performance of hardware processing units with hybrid custom floating-point approximation.

Hardware mapping		Computation schedule (ms)			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.307	0.362
H1_CONV	Conv1	0.057	0.654	1.309	2.020
H2_POOL	Pool1	0.713	0.131	1.098	1.942
	Pool2	0.845	0.125	1.098	2.068
H3_CONV	Conv2	0.972	0.285	1.199	2.456
	Conv3	1.258	0.279	1.184	2.721
H4_POOL	Pool3	1.538	0.037	0.484	2.059
H5_FC	FC	1.577	0.091	0.438	2.106
HY_OUT	CPU	1.669	0.004	0	1.673

TABLE 7. Performance of hardware processing units with hybrid logarithmic approximation.

Hardware mapping		Computation schedule (ms)			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.264	0.319
H1_CONV	Conv1	0.057	0.655	1.271	1.983
H2_POOL	Pool1	0.714	0.130	1.074	1.918
	Pool2	0.845	0.126	1.106	2.077
H3_CONV	Conv2	0.973	0.285	1.179	2.437
	Conv3	1.258	0.278	1.176	2.712
H4_POOL	Pool3	1.538	0.037	0.488	2.063
H5_FC	FC	1.577	0.091	0.388	2.056
HY_OUT	CPU	1.669	0.004	0	1.673

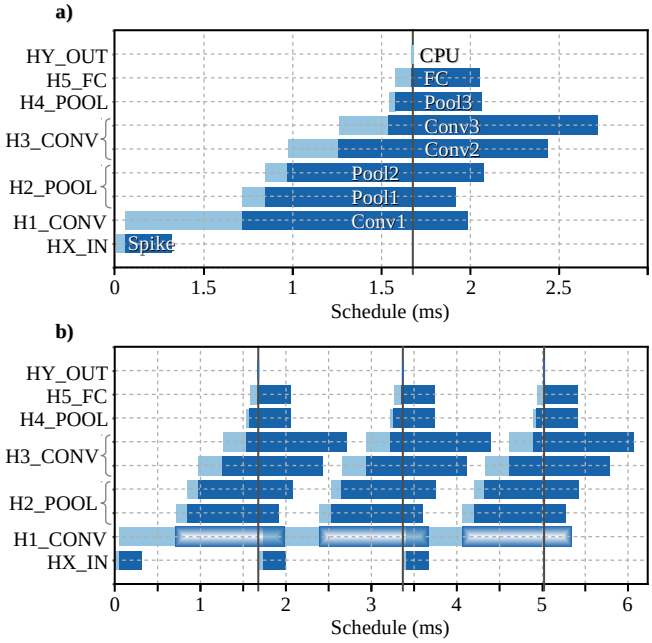

FIGURE 13. Performance of processing units with hybrid logarithmic approximation, (a) exhibits computation schedule, and (b) illustrates cyclic computation schedule.

in **Tab.** 7 and **Fig.** 13. As shown in the computation schedule in **Tab.** 7 and **Fig.** 13, this implementation achieves a peak acceleration of 55X and a power efficiency of 5.5 GFLOPS/s/W. This quality configuration achieves an accuracy degradation 0.84% on correct regressions on the 500 validation samples.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 8.

TABLE 8. Resource utilization and power dissipation of processing units with hybrid logarithmic approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,086	4,804	19	21	78
FC	3,046	4,873	8	8	66

C. RESULTS AND DISCUSSION

As a reference, the inference on embedded CPU using standard 32-bit floating-point achieves an accuracy gain of 0.3% with a latency of 3,450.28ms. As a second reference point, the inference on TP with standard floating-point presents a latency of 34.5ms, as result we get a 10.7× latency enhancement.

As a demonstration of the proposed hardware/software architecture, the inference with TP using 5-bit custom floating-point (4-bit exponent, 1-bit mantissa) and 4-bit logarithmic (4-bit exponent) achieves 55.5× latency enhancement. This results in an accuracy gain of 0.33% and degradation of 0.46%, respectively.

Regarding resource utilization and power dissipation, the TP with 5-bit custom floating-point has a 43.24% reduction of BRAM, and a 12.35% of improvement in energy efficiency over the standard floating-point implementation. However, the hybrid dot-product with custom floating-point does not reuse the available floating-point operator cores instantiated from other computational sections (see **Tab.** 4). Therefore, the logic required for the dot-product must be implemented, which is reflected as additional utilization of LUT and FF resources. The experimental results of the design exploration are summarized in **Tab.** 9. The platform implementations are summarized in **Tab.** 10, and their power dissipation breakdowns are presented in **Fig.** 14.

D. 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.** 11, 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× in model A at the tensor operation *(3A) Conv* with a power reduction

TABLE 9. Experimental results.

Dot-product implementation	PU	Post-implementation resource utilization				Power (mW)	Latency		Accuracy (%)[e]	
		LUT	FF	DSP	BRAM 18K		T_{SC} (ms)	Gain[d]	Noise 0%	50%
Standard floating-point computation[a]	Conv	2,765	4,366	19	37	89	3.18	10.7x	98.98	98.63
	FC	2,649	4,189	8	9	66				
Hybrid custom floating-point approx[b]	Conv	3,139	4,850	19	25	82	1.67	20.5x	98.97	98.47
	FC	3,265	5,188	8	9	66				
Hybrid logarithmic approximation[c]	Conv	3,086	4,804	19	21	78	1.67	20.5x	98.84	95.22
	FC	3,046	4,873	8	8	66				

[a] Reference with standard floating-point arithmetic (IEEE 754).
[b] Synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[c] Synaptic weight with number representation composed of 4-bit exponent.
[d] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[e] Accuracy on 10,000 image test set with 1000 spikes.

TABLE 10. Platform implementations.

Platform implementation	Post-implementation resource utilization				Power (W)	Clock (MHz)	Latency		Accuracy (%)[f]
	LUT	FF	DSP	BRAM 18K			T_{SC} (ms)	Gain[e]	
Ref. [?][a]	42,740	57,118	49	92	2.519	250	4.65	7.4x	99.02
This work (standard floating-point computation)[b]	39,514	56,036	82	180	2.420	200	3.18	10.7x	98.98
This work (hybrid custom floating-point approx)[c]	42,021	58,759	82	156	2.369	200	1.67	20.5x	98.97
This work (hybrid logarithmic approximation)[d]	41,060	57,862	82	148	2.324	200	1.67	20.5x	98.84

[a] Reference architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754).
[b] Reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754).
[c] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[d] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent.
[e] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[f] Accuracy on 10,000 image test set with 1000 spikes.

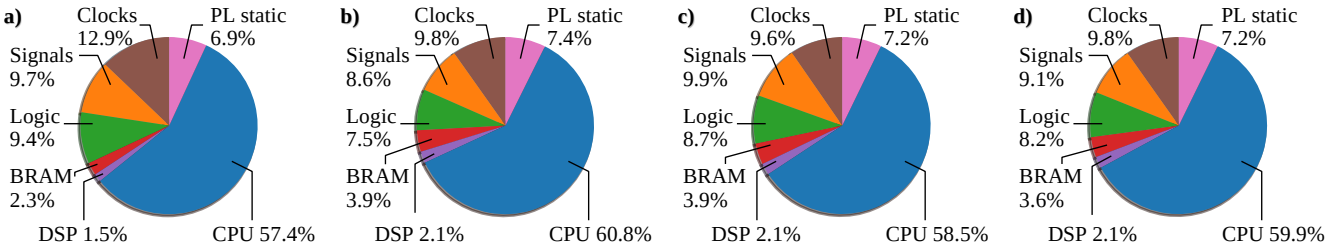

FIGURE 14. Power dissipation breakdown of platform implementations, (a) **Ref. [?]** architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754), (b) reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754), (c) proposed architecture with hybrid custom floating-point approximation, and (d) proposed architecture with hybrid logarithmic approximation.

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.

TABLE 11. 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

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 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. This approach is fundamentally more efficient compared to equivalent fixed-point number representations. 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 . . . , 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] C. Wu, M. Wang, X. Chu, K. Wang, and L. He, "Low-precision floating-point arithmetic for high-performance fpga-based cnn acceleration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–21, 2021.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[16] C. Mei, Z. Liu, Y. Niu, X. Ji, W. Zhou, and D. Wang, "A 200mhz 202.4 gflops@ 10.8 w vgg16 accelerator in xilinx vx690t," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 784–788.

[17] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in neural information processing systems*, vol. 28, 2015.

[19] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[20] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using intel® fpgas," in *2018 IEEE 26th annual international symposium on field-programmable custom computing machines (FCCM)*. IEEE, 2018, pp. 73–80.

[21] 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.

[22] 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.

[23] 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.

[24] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *arXiv preprint arXiv:1703.03073*, 2017.

[25] S. O. Settle, M. Bollavaram, P. D'Alberto, E. Delaye, O. Fernandez, N. Fraser, A. Ng, A. Sirasao, and M. Wu, "Quantizing convolutional neural networks for low-power high-throughput inference engines," *arXiv preprint arXiv:1805.07941*, 2018.

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

[27] 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.

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

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

[30] 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.