IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

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 sensor analytics 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, improving efficiency of AI hardware engines must be considered. In this paper, we present the Hybrid-Float6 (HF6) quantization on shallow CNNs for sensor data analytics and its dedicated hardware design. This approach reduces over-fit on feature extraction and improves generalization. As dedicated hardware design, we propose a fully customizable tensor processor (TP) implementing a pipelined vector dot-product with HF6. This approach reduces energy consumption and resource utilization. The proposed embedded hardware/software architecture is unified with TensorFlow Lite. We evaluate the applicability of our approach with a data analytics 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.

INDEX TERMS Convolutional neural networks, structural health monitoring, hardware accelerator, TensorFlow Lite, embedded systems, FPGA, custom floating-point

I. INTRODUCTION

THERE is a growing demand for ubiquitous AI sensor data analytics. Industry 4.0 and smart city infrastructure leverage AI solutions to increase productivity and adaptability [1]. These solutions are powered by advances in machine learning (ML), hardware engines, and big data. Hence, enhancement of these should be considered for research, as they are the machinery of the future.

CNN-based models represent the essential building blocks in 2D pattern recognition tasks. Sensor-based applications such as mechanical fault diagnosis [2], [3], structural health monitoring (SHM) [4], human activity recognition (HAR) [5], hazardous gas detection [6] have been powered by CNN models in industry and academia.

CNN-based models, as one of the main types of artificial neural networks (ANNs), have been widely used in sensor analytics with automatic learning from sensor data [7]–[10]. In this context, CNN models are applied for automatic feature learning, usually, from 1D time series as well as for 2D time-frequency spectrograms. CNN models provide advantages such as local dependency, scale invariance, and noise resilience in data analytics. However, these models represent compute-intensive and power-hungry tasks, particularly, for low-power and resource-limited Internet-of-Things (IoT) devices.

Dedicated hardware architectures are typically used to enhance compute performance and 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) [11]. 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 compute costly CNN models such as AlexNet, VGG-16, and ResNet-18. The power supply demands, physical dimensions, air cooling and heat sink requirements, and in some cases their elevated price make these implementations unsustainable and not always feasible for ubiquitous resource-constrained applications.

Furthermore, to reduce the computational cost for CNN inference there are two types of research [12]: the first one is deep compression including weight pruning, weight

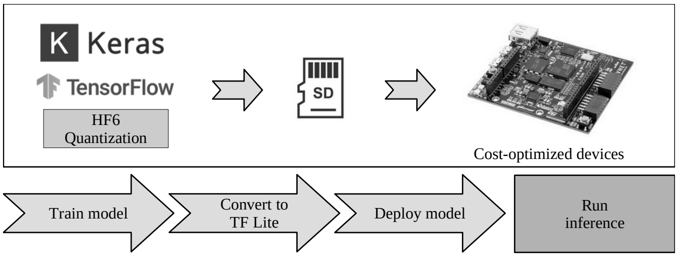

FIGURE 1. The workflow of our approach on embedded FPGAs.

quantization, and compression storage [13], [14]; 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 8-bit floating-point computation have been proposed [12], [15], [16]. However, these implementations are inadequate for embedded applications, since the target devices are high-end FPGA and PCIe architectures.

The aforementioned works have good accuracy with retraining, more aggressive data representations such as binary [17], ternary [18], and mixed precision (2-bit activations and ternary weights) [19] may suffer from great accuracy loss even with time-consuming retraining. The afforded mentioned limitations make these implementations inadequate for accurate data analytics in low-power embedded applications.

In this paper, we present the Hybrid-Float6 quantization on CNNs for sensor data analytics and its dedicated hardware design on low-power resource-constrained embedded FPGAs. The HF6 implements 6-bit floating-point quantization on the trainable parameters of convolution layers and standard floating-point on feature maps. This approach reduces over-fit on feature extraction and improves generalization. As dedicated hardware design, we propose a parameterized tensor processor implementing a pipelined vector dot-product with HF6. This approach reduces energy consumption and resource utilization facilitating on-chip stationary weights on limited footprint devices. The embedded hardware/software architecture is unified with TensorFlow Lite using delegate interface to accelerate *Conv2D* tensor operations. We evaluate the applicability of our approach with a CNN model and hardware design exploration for sensor analytics of SHM for anomaly localization based on regression. The embedded hardware/software framework is demonstrated on XC7Z007S as the smallest and most inexpensive Zynq SoC device, see **Fig.** 1. To the best of our knowledge, this is the first research addressing 6-bit floating-point quantization on CNN-based models and its dedicated hardware implementation.

Our main contributions are as follows:

1) We present the Hybrid-Float6 quantization. This approach improves generalization by reducing over-fit on feature extraction. The HF6 is wrapped into the standard floating-point representation (IEEE 754) allowing compatibility with standard hardware. Therefore, it can be beneficial for inference in other devices.
2) We develop a hardware/software co-design framework for sensor analytics applications on low-power and resource-limited FPGAs. This is a scalable architecture integrating TensorFlow Lite core in the embedded system.
3) We present a customizable tensor processor as a dedicated hardware for HF6. This design computes *Conv2D* tensor operations employing a pipelined vector dot-product with parametrized on-chip memory utilization. The compute engine of the tensor processor is implemented with standard floating-point, fixed-point, and HF6.
4) We demonstrate the potential of our approach by addressing CNN model and hardware design exploration for sensor analytics of anomaly localization based on regression for SHM. We evaluate inference accuracy, compute performance, hardware resource utilization, and energy consumption.

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 [20], 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 [21], 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 [22] proposed a mixed data representation with floating-point for weights and fixed-point for activations (e.g., outputs of a layer). Reference [23] developed an 8-bit floating-point quantization scheme, which needs an extra inference batch to compensate for the quantization error. However, Reference [22] and Reference [23] did not present a circuit design for their approaches.

1) FPGA implementations

Reference [15] 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 [12]. Another 8-bit arithmetic, called block floating-point (BFP), is also applied [16], 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 [24]. We denote *Conv* as *Conv2D* operator.

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

IV. SYSTEM DESIGN

The system design is a hardware/software co-design framework for low-power AI deployment. This architecture allows design exploration of dedicated hardware for TensorFlow Lite on low-cost embedded FPGAs.

A. BASE EMBEDDED SYSTEM ARCHITECTURE

The base embedded system architecture implements a co-operative hardware-software platform. **Fig.** 2 illustrates the top-level hardware architecture. The TPs execute low-level tensor operations delegated from the embedded CPU. The TPs employ AXI-Lite interface for configuration and AXI-Stream interfaces via Direct Memory Access (DMA) for data movement from DDR memory. Each TP asserts an interrupt flag once the job/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 and energy consumption by customizing the TPs prior to the hardware synthesis.

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 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.

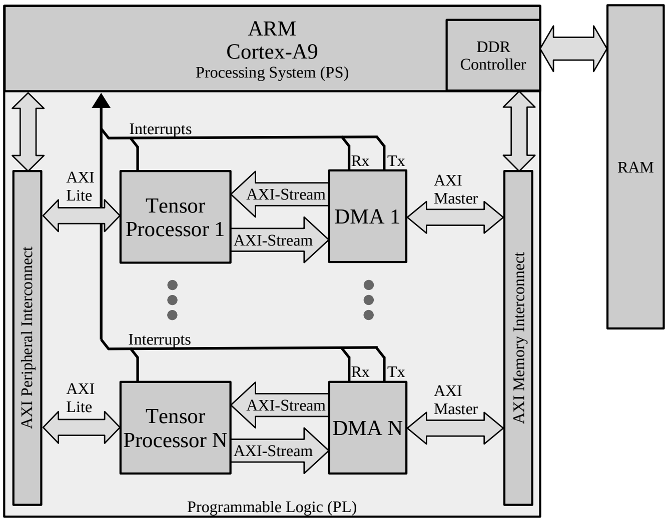

FIGURE 2. Base embedded system architecture.

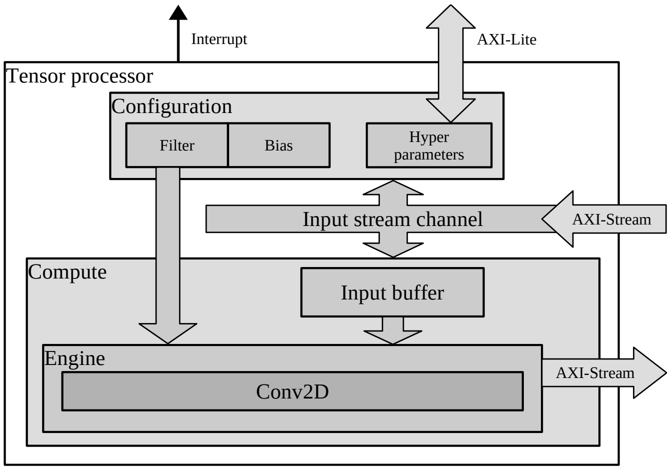

FIGURE 3. Hardware architecture of the proposed tensor processor.

The TP is an extensible hardware module that executes low-level tensor operations. In this paper, we focus on the *Conv2D* tensor operator that executes inference of convolution layers.

1) Modes of operation

The TP has two modes of operation: *configuration* and *execution*.

- In *configuration* mode, the TP receives the tensor operation ID for *Conv2D* 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, which are locally stored in BRAM.
- In *execution* mode, the TP executes the tensor operation according to the hyperparameters given in the configuration mode. During execution, the input and output tensors are moved from/to the DDR memory via DMA.

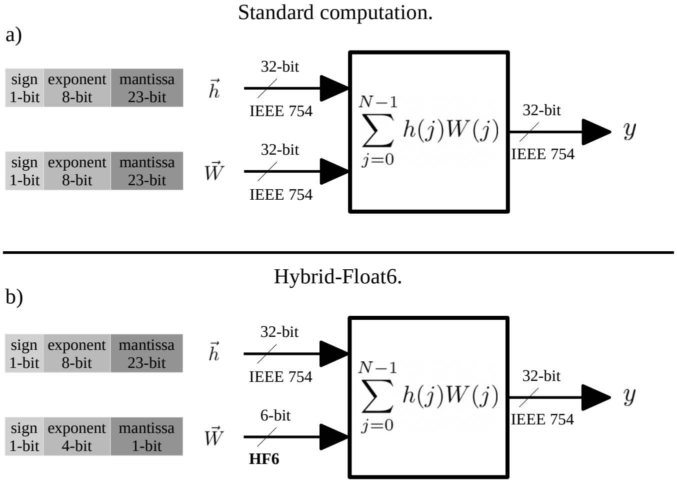

FIGURE 4. Dot-product hardware module with (a) standard floating-point and (b) Hybrid-Float6.

2) Dot-product with hybrid floating-point

We implement the floating-point computation adopting the dot-product with hybrid custom floating-point [25]. 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.** (2), where N is the 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 results. Both the exponent and mantissa bit widths of the filter and bias buffers are set to a 4-bit exponent and a 1-bit mantissa (E4M1), which corresponds to float6 quantization.

$$L_{hf} = N + 7 \tag{2}$$

3) On-chip memory utilization

The total on-chip memory utilization on the TP is defined by **Eq.** (3), 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 the design. The on-chip memory buffers are defined in bits. **Fig.** 6 illustrates the convolution operation utilizing the on-chip memory buffers.

$$TP_M = Input_M + Filter_M + Bias_M + V_M \tag{3}$$

The memory utilization of *input buffer* is defined by **Eq.** (4), 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

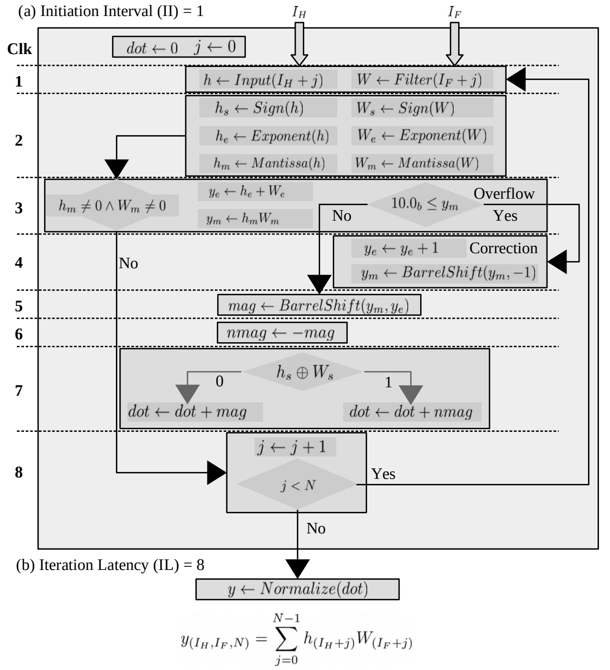

FIGURE 5. Pipelined hardware module for vector dot-product with hybrid custom floating-point, (a) exhibits the initiation interval of 1 clock cycle, and (b) presents the iteration latency of 8 clock cycles. I_H and I_F represent the input and filter buffer indexes, respectively.

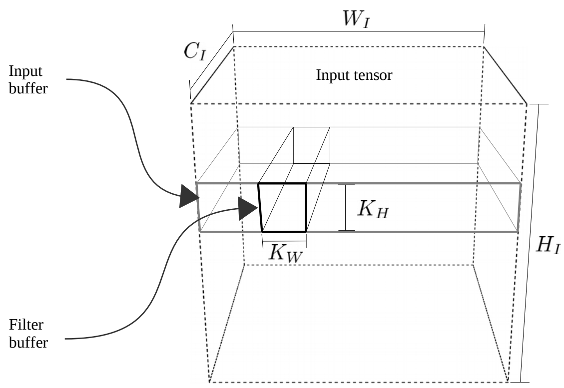

FIGURE 6. Design parameters for on-chip memory buffers on the TP.

channels, and $BitSize_I$ is the bit size of each input tensor element.

$$Input_M = K_H W_I C_I BitSize_I \tag{4}$$

The memory utilization of *filter buffer* is defined by **Eq.** (5), 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.

$$Filter_M = C_I K_W K_H C_O BitSize_F \tag{5}$$

The memory utilization of *bias buffer* is defined by **Eq.** (6), where C_O is the number of output channels, and $BitSize_B$ is the bit size of each bias element.

$$Bias_M = C_O BitSize_B \tag{6}$$

As a design trade-off, **Eq.** (7) defines the capacity of output channels based on the given design parameters. The total on-chip memory TP_M determines the TP capacity.

$$C_O = \frac{TP_M - V_M - K_H W_I C_I BitSize_I}{C_I K_W K_H BitSize_F + BitSize_B} \tag{7}$$

The floating-point formats implemented in the TP are defined by $BitSize_F$, $BitSize_B$ and $BitSize_I$. The HF6 defines 6-bits for $BitSize_F$ and $BitSize_B$, and 32-bits for $BitSize_I$. These are design parameters defined before hardware synthesis. This allows fine control of BRAM utilization, which is suitable for resource-limited devices.

C. TRAINING METHOD

The CNN-regression models are trained and quantized in separate stages.

1) Training with Iterative Early Stop Cycle

To achieve better performance on CNN-regression models, we implement a training procedure with iterative early stop cycle. This allows to reach better local minima. This is a four steps process: (1) a base model is obtained with an initial training, (2) the base model is iteratively re-trained to search for better local minima, (3) in case of a better local minimum, the base model is updated and used for subsequent re-training iterations, (4) the training cycle is automatically aborted with a patience of unsatisfactory search iterations. See **Algorithm** 1. This method minimizes the MSE, which is calculated with the Euclidean distance between the validation and predicted coordinates. The early stop has a patience of 10 epochs and mini-batch size between 512 to 1024 samples.

2) Quantized Aware Training

The quantization method is integrated into the training process, this operates after each mini-batch update. The quantization is applied on the trainable parameters of convolution layers. This method is implemented as a callback function in the TensorFlow/Keras framework, see **Algorithm** 2.

The quantization method uses rounding strategy to reduce the floating-point memory representation. This maps the full precision floating-point values to the closest representable 6-bit floating-point values, see **Algorithm** 3. We have observed that the exponent bit size plays a more predominant influence on the model accuracy than the mantissa bit size. In [22], Lai et al. demonstrated that 4-bit exponent is adequate and consistent across different networks (SqueezeNet, AlexNet, GoogLeNet, VGG-16). In this work, we investigate 4-bit exponent and 1-bit mantissa. This method quantizes the filter and bias tensors of the convolution layers.

The resulting quantized parameters are wrapped into the standard floating-point. This allows compatibility with standard hardware. On the proposed embedded system, the 6-bit floating-point values are extracted and buffered in the on-chip memory of the TP during operation.

Algorithm 1: Training with iterative early stop cycle.

input: $MODEL$ as the input model.
input: D_{train} as the training data set.
input: D_{val} as the validation data set.
input: N_I as the stop patience for iterative training cycle.
input: N_E as the early stop patience (epochs) for training.
input: B_{size} as the mini-batch size.
output: $MODEL$ as the full-precision output model.
  $Train(MODEL, D_{train}, D_{val}, N_E, B_{size})$
  $mse_i \leftarrow Evaluate(MODEL, D_{val})$ // Benchmark
  $n_i \leftarrow 0$
  while $n_I < N_I$ **do**
    // Iterative early stop cycle
    $Train(MODEL, D_{train}, D_{val}, N_E, B_{size})$
    $mse_v \leftarrow Evaluate(MODEL, D_{val})$
    if $mse_v < mse_i$ **then**
      $Update(MODEL)$
      $mse_i \leftarrow mse_v$
    else
      $MODEL \leftarrow LoadPreviousModel()$
      $n_I \leftarrow n_I + 1$
    end if
  end while

Algorithm 2: Quantization aware training.

input: $MODEL$ as the full-precision input model.
input: E_{size} as the target exponent bits 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: N_{ep} as the number of epochs.
input: B_{size} as the mini-batch size.
output: $MODEL$ as the quantized output model.
  $MODEL \leftarrow Quantize(MODEL, E_{size}, M_{size})$
  $mse_i \leftarrow Evaluate(MODEL, D_{val})$ // Benchmark
  $Train(MODEL, D_{train}, D_{val}, N_{ep}, B_{size})$

  OnMiniBatchUpdate_Callback():
  $MODEL \leftarrow Quantize(MODEL, E_{size}, M_{size})$
  if $1 < epoch$ **then**
    // Update model after first epoch
    $mse_v \leftarrow Evaluate(MODEL, D_{val})$
    if $mse_v < mse_i$ **then**
      $Update(MODEL)$
      $mse_i \leftarrow mse_v$
    end if
  end if

Algorithm 3: 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

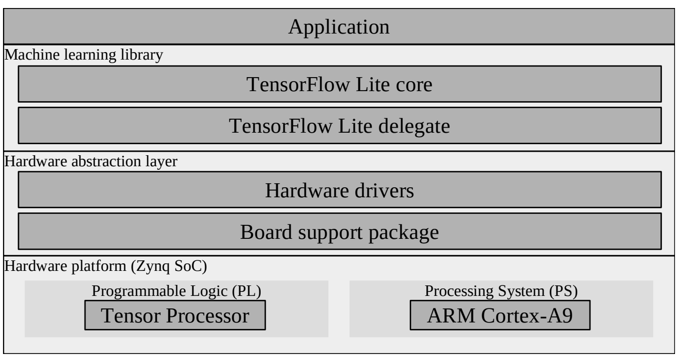

FIGURE 7. Base embedded software architecture.

D. EMBEDDED SOFTWARE ARCHITECTURE

The software architecture is a layered object-oriented application framework written in C++, see **Fig.** 7. A description of the software layers is as follows:

- *Application*: As the highest level of abstraction, this software layer implements the analytics application using the ML library.
- *Machine learning library*: This layer consist of TensorFlow Lite for micro controllers. This offers a comprehensive high level API that allows ML inference. This provides delegate software 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

In this section, we present experimental results of the HF6 concept on low-cost sensor analytics applications. As a use case, we present a CNN-regression model to predict x- y-coordinates of structural anomalies based on acoustic sensor data. We compare quantitative and qualitative aspects of the data analytics using floating-point, fixed-point, and HF6 approach.

To demonstrate the HF6 hardware concept, we deploy the CNN model for low-power inference in the smallest Zynq SoC. We compare the performance of the dedicated hardware implemented with standard floating-point and HF6 architecture.

A. SENSOR ANALYTICS APPLICATION

The sensor analytics model is designed to predict x- y-coordinates of acoustic emissions on a metal plate with noise disturbance. We present the experimental setup, data sets, and the CNN-regression model.

1) Experimental Setup

The experiment uses eight piezoelectric sensors (Vallen Systeme VS900) fixed on a metal plate (90 cm x 86.6 cm x 0.3 cm). The VS900 devices can operate either in active or passive mode. Six VS900 are used in passive mode as acoustic sensors and two in active mode to produce acoustic emissions. These acoustic emissions represent the anomalies on x- y- coordinates and the noise disturbance of the system. See **Fig.** 9(a). The acoustic emissions are labeled with their coordinates to create data sets.

2) Data Sets

The data sets are recorded with pulses and the x- y- coordinates as labels. The pulses for training and validation data sets are shown in **Fig.** 9(b) and **Fig.** 9(c), respectively. The pulses for training and validation data sets are mutually exclusive, this exclusion is represented by the cross symbols in **Fig.** 9(c).

In order to create a reproducible acoustic emission, narrow-banded in the frequency domain, we use 9-cycle sine pulse in a Hanning window with central frequency f_c. We assume guided Lamb waves based on the plate structure. The narrow-band behavior also reduces the dispersion of the acoustic emission waves [26]. The waveform can be expressed as a function of time t as follows:

$$x_{\mathrm{pulse}}(t) = \frac{1}{2}\Big(1 - \cos\frac{f_{\mathrm{c}}t}{5}\Big)A_0 \sin f_{\mathrm{c}}t. \tag{8}$$

In order to generate the data sets, we use slightly different pulse amplitudes and frequencies for excitation. The pulse frequency f_c is varied in 1 kHz steps between 300 kHz and 349 kHz and the amplitude A_0 is varied in 0.1 V steps between 2.6 V and 3.5 V. This results in 500 different pulses for each of the excitation points. **Fig.** 9(c) illustrates the grid layout used to collect samples for the data sets. This grid is 10 by 10 and it does not consider the four corners as they are used for structure holders.

The labeled pulses and the noise disturbance signals are generated by arbitrary waveform generators (AWGs). The sensor signals are recorded via a Vallen AMSY-6 measurement system with a resolution of 18 bits and a sampling rate of $f_{\mathrm{S}} = 10MHz$. The noise disturbance is a gaussian noise signal with amplitudes between 0-3 V. This noise is applied via the piezoelectric device N at $x = 0.227$ and $y = 0.828$, see **Fig.** 9(a).

In order to obtain both time and frequency domain information, the acquired pulses are converted into the time-frequency domain using the Short-Time Fourier Transform (STFT). It is calculated as follows [27]:

$$\mathcal{F}^{\gamma}_{m,k} = \sum_{n=0}^{N-1} x[n]\cdot\gamma^*[n - m\Delta M]\cdot \mathrm{e}^{\frac{-j2\pi kn}{N}} \tag{9}$$

Here $x[n]$ describes a discrete-time signal and $\gamma^*[n - m\Delta M]\cdot \mathrm{e}^{\frac{-j2\pi kn}{N}}$ the time- and frequency-shifted window function inside the considered interval $[0, N-1]$. ΔM describes the time shift and N the transformation window. Since only discrete frequencies and time points are considered, $m = 0, 1, ..., M-1$ is valid. This complex-valued STFT is converted to real numbers via the magnitude square for pictorial representation in a spectrogram $\mathcal{S}_{m,k}$:

$$\mathcal{S}_{m,k} = \left|\mathcal{F}^{\gamma}_{m,k}\right|^2 = \left|\sum_{n=0}^{N-1} x[n]\cdot\gamma^*[n - m\Delta M]\cdot \mathrm{e}^{\frac{-j2\pi kn}{N}}\right|^2 \tag{10}$$

In addition, the spectrograms used are scaled in decibels. The spectrogram in decibels $\mathcal{S}_{m,k,\mathrm{dB}}$ results in $\mathcal{S}_{m,k,\mathrm{dB}} = 20\cdot\log_{10}(\mathcal{S}_{m,k})$. For the conversion of the data a signal length of 400 µs (75 µs pretrigger and 325 µs post trigger) is used. Thus, the arrival times of the pulses are included in the spectrogram for all channels and labeled positions. A Blackman window function [28], a Fast Fourier Transform (FFT) length of 32 samples, and an overlap of 8 samples are used. The spectrograms are calculated for frequencies in the range of 100 kHz to 500 kHz. This results in a spectrogram with 8x16 values (8 frequency values, 16 time values). In addition to the original 400 µs windows, four further variants with time shifts of 15 µs/ 30 µs/ 45 µs/ 60 µs were calculated in order to generate larger data sets. Subsequently, all spectrograms were converted to grayscale with scaling between -100dB and -40dB, see **Fig.** 8. Overall, the data set has a size of 500 (pulses) · 5 (spectrograms) · 6 (listening sensors) · 96 (excitation points) = 1,440,000 images.

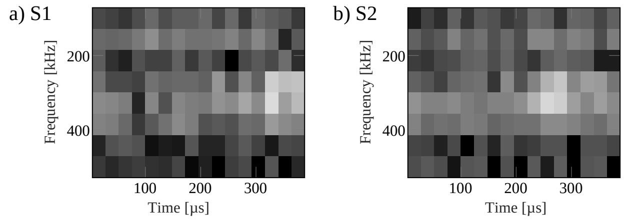

FIGURE 8. Spectrograms of sensors S_1, S_2 converted to grayscale for pulses at $x = 0.105$, $y = 0.109$ with noise disturbance.

3) CNN-Regression Model

The data analytics is implemented with a CNN-regression model, see **Fig.** 10. The structure of the model is described below:

a) Input patterns. This is an input tensor composed of spectrograms from the sensor signals. The tensor shape is defined by $S \times T \times F$, where S is the number of sensors, and $T \times F$ is the time-frequency resolution of the spectrograms, see **Fig.** 10(a).
b) Feature extraction. This is composed of three blocks of convolution, batch normalization, and max-pooling layers, see **Fig.** 10(b). The number of channels in the convolution layers are defined by the hyper-parameters A, B, and C.
c) Regression function. This is an arbitrary function implemented with two fully connected layers and an output layer with linear activation, see **Fig.** 10(c).

B. TRAINING

1) Base Model

The model in **Fig.** 10 is trained using Adam algorithm with iterative early stop, described in **Algorithm** 1. The Adam optimizer is configured with the default settings presented in [29]: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1\mathrm{e}{-8}$. The training cycle patience is 10 search iterations, the optimizer is executed with early stop patience of 10 epochs, and mini-batch size of 512 samples. ($N_I = 10$, $N_E = 10$, $B_{size} = 512$.)

The training results are illustrated in **Fig.** 11(a). The initial model is obtained at the first early stop. Each stop initializes the moving averages of the Adam optimizer. In subsequent iterations, the model is updated when better minimum is reached.

The model achieves $MSE = 0.0122m^2$ and $MAE = 0.0955m$. See **Fig.** 12(a). In total, the training takes 379 epochs in 25 stops. The first stop takes 43 epochs for the initial model and subsequent training iterations take an average of 14 epochs. The training time is 216 minutes using a

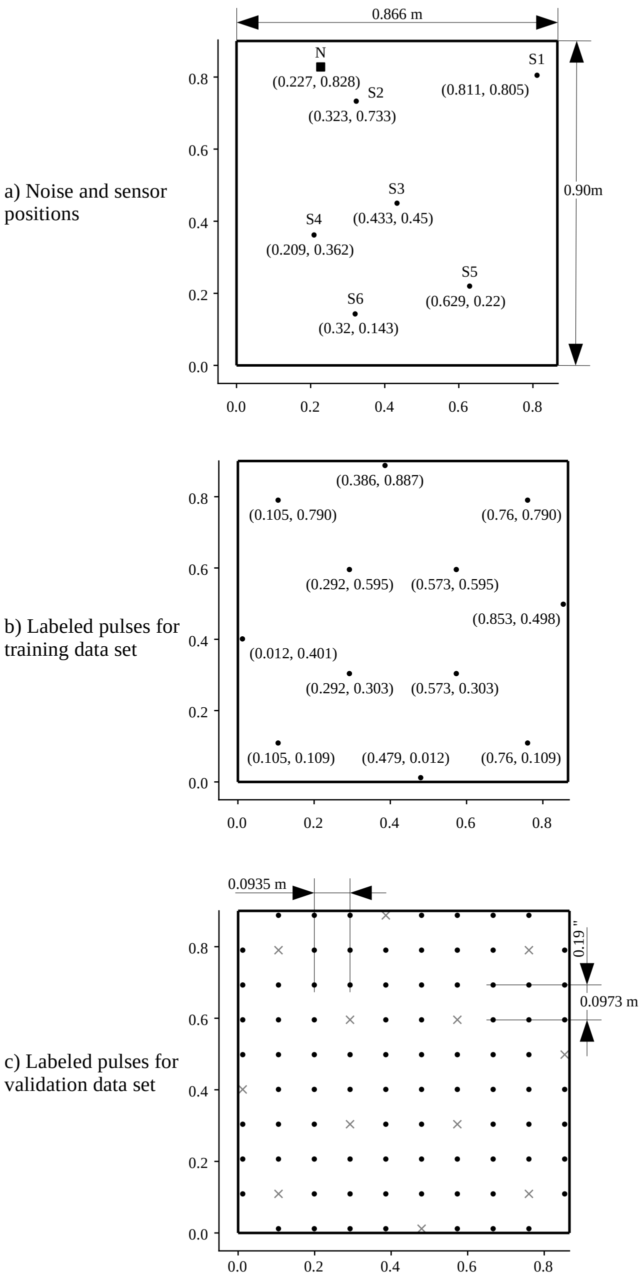

FIGURE 9. (a) Illustrates the Sampling layout for training and validation data set.

PC with AMD Ryzen 5 5600H and NVIDIA GeForce RTX 3050.

2) TensorFlow Lite 8-bit quantization

This integer quantization is an optimization method that converts 32-bit floating-point numbers (such as weights and activations) to the nearest 8-bit fixed-point numbers. This quantization scheme allows inference to be carried out using integer-only arithmetic [26].

The base model is quantized using the TensorFlow Lite library with integer-only quantization settings. The filter and bias tensors are represented by 8-bit and 32-bit signed integers, respectively. The input and output activation tensors are represented by 8-bit signed integer. For convolution layers, this quantization includes two additional tensor coefficients (output-multiplier and output-shift). These tensors are the same shape as the bias tensor and are represented by 32-bit signed integers.

The model achieves $MSE = 0.0122m^2$ and $MAE = 0.0952m$. See **Fig.** 12(b). The MAE gets an improvement of 0.3% compared to the base model. We attribute this to the regularization effect.

3) Quantization Aware Training for HF6

The HF6 quantization is a post-training step. This is applied with the quantization aware training method described in **Algorithm** 2. We run this method during two epochs with mini-batch size of 10 samples with quantization parameters of 4-bit exponent and 1-bit mantissa. ($N_{ep} = 2$, $B_{size} = 10$, $E_{size} = 4$, $M_{size} = 1$.)

The quantization aware training is illustrated in **Fig.** 11(b). Initially, the model is quantized with HF6 format without optimization, this obtains $MSE = 0.0188m^2$ and $MAE = 0.1232m$, these correspond to 54% and 29% error increase, respectively. This represents the inference of the base model without quantization aware training on HF6 hardware. See **Fig.** 12(c).

The final model after the quantization aware training achieves $MSE = 0.0112$ and $MAE = 0.0919$, these correspond to 9% and 3.5% error reduction, respectively. See **Fig.** 12(d).

4) Quantization Aware Training for Hybrid-Logarithmic 6-bit

For comparison, we quantize and optimize the base model with 6-bit logarithmic quantization on convolution layers. The filter and bias tensors are represented by 6-bit signed logarithmic. The input and output activation tensors are represented by standard floating-point. This is applied with **Algorithm** 2. ($N_{ep} = 2$, $B_{size} = 10$, $E_{size} = 5$, $M_{size} = 0$.)

The model after the quantization aware training achieves $MSE = 0.0123$ and $MAE = 0.0968$, these correspond to 0.8% and 1.36% error increase, respectively. See **Fig.** 12(e).

C. HARDWARE DESIGN EXPLORATION

The proposed hardware/software co-design is demonstrated on the Zynq-7007S system-on-chip (SoC) in the MiniZed development board. 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 [30]. 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. ??**. 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

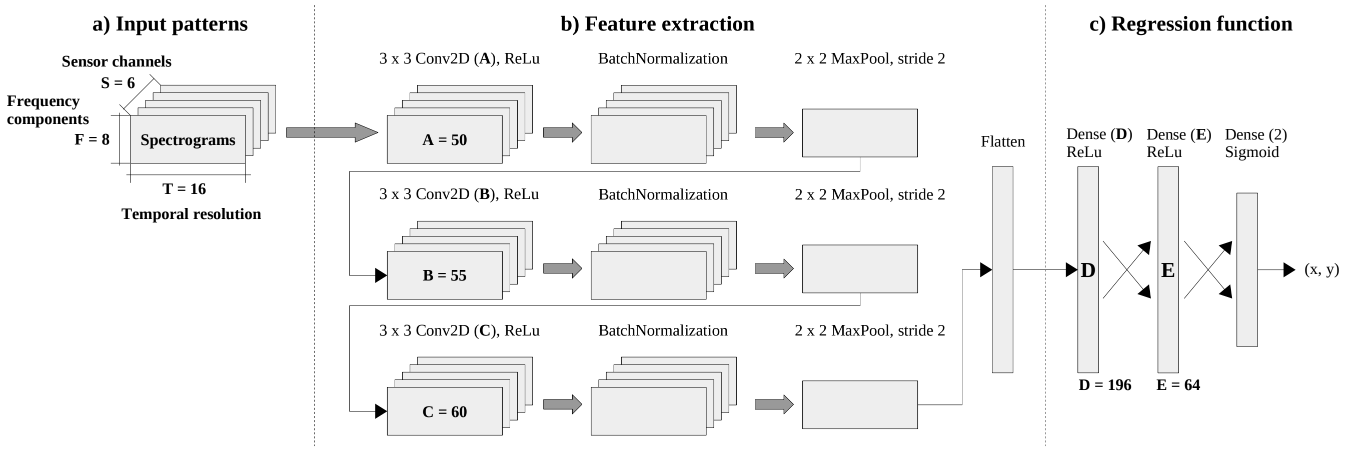

FIGURE 10. CNN model for sensor analytics.

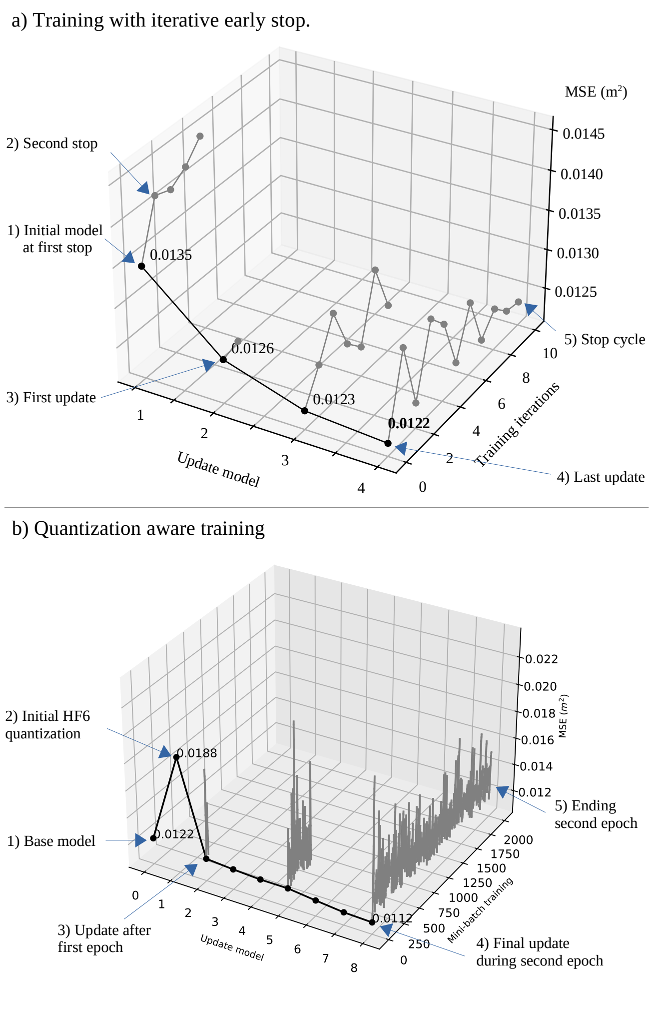

FIGURE 11. Training results.

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.

D. 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 schedule of the CNN inference are displayed in **Tab. ??** and **Fig. ??** 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. ??** and **Fig. ??**.

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

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 ac-

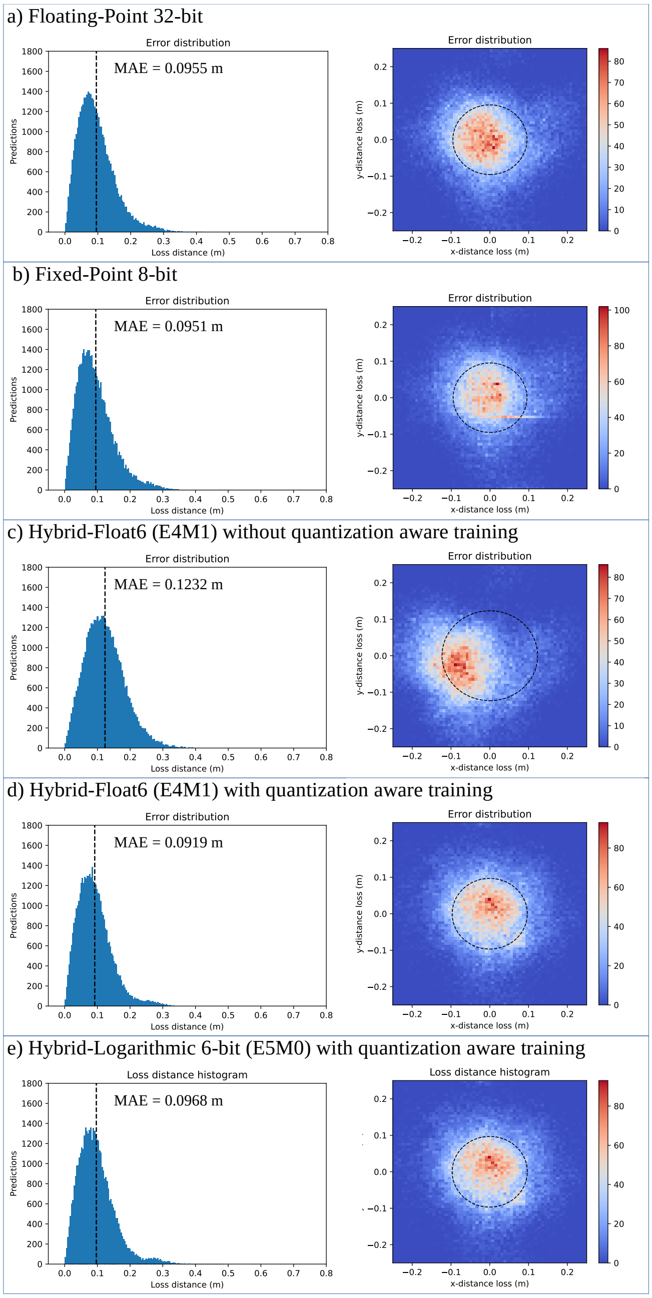

FIGURE 12. Performance of the model with different quantizations for quantitative and qualitative comparison.

complishes floating-point arithmetic operations by mapping them onto Xilinx LogiCORE IP cores, these floating-point operator cores are instantiated in the resultant RTL [31]. 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.** 1.

TABLE 1. 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

TABLE 2. Performance of the CPU and TP on each CONV_2D tensor operation, computational cost, latency, throughput, power efficiency, and energy consumption.

Operation	MFLOP	t (ms)	MFLOP/s	MFLOP/s/W	EDP (mJ)
		a) CPU @666MHz, 1.187 W			
(1) CONV_2D	0.691	112.24	6.16	5.19	133.23
(2) CONV_2D	1.584	213.13	7.43	6.26	252.99
(3) CONV_2D	0.475	46.59	10.20	8.59	55.31
		b) TP Floating-Point @200MHz, 85 mW			
(1) CONV_2D	0.691	12.49	55.34	651.11	1.06
(2) CONV_2D	1.584	16.39	96.66	1,137.20	1.39
(3) CONV_2D	0.475	3.59	132.44	1,558.13	0.30
		c) TP Hybrid-Float6 @200MHz, 84 mW			
(1) CONV_2D	0.691	6.92	99.81	1,188.24	0.58
(2) CONV_2D	1.584	4.41	358.94	4,273.09	0.37
(3) CONV_2D	0.475	0.99	482.44	5,743.29	0.08

TABLE 3. Resource utilization and power dissipation on the Zynq-7007S SoC.

Platform	Post-implementation resource utilization				Power (W)
	LUT	FF	DSP	BRAM 36Kb	
Floating-Point	5,578 39%	8,942 31%	23 35%	41.5 83%	1.429
Hybrid-Float6	7,313 51%	10,330 36%	20 30%	15 30%	1.424

E. 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 presented in **Fig.** ??. 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

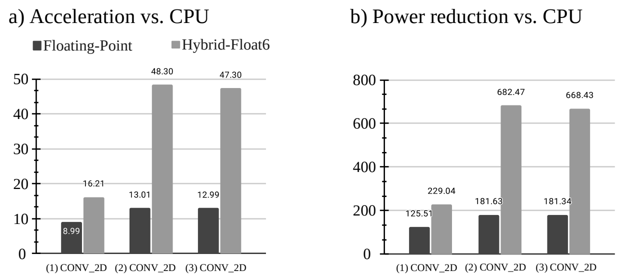

FIGURE 13. Acceleration and power reduction of the TP with floating-point and HF6 vs. CPU on the Zynq-7007S SoC.

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. ??** and **Fig. ??**.

As shown in the computation schedule in **Tab. ??** and **Fig. ??**, 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. ??**.

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 in **Tab. ??** and **Fig. ??**. As shown in the computation schedule in **Tab. ??** and **Fig. ??**, 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. ??**.

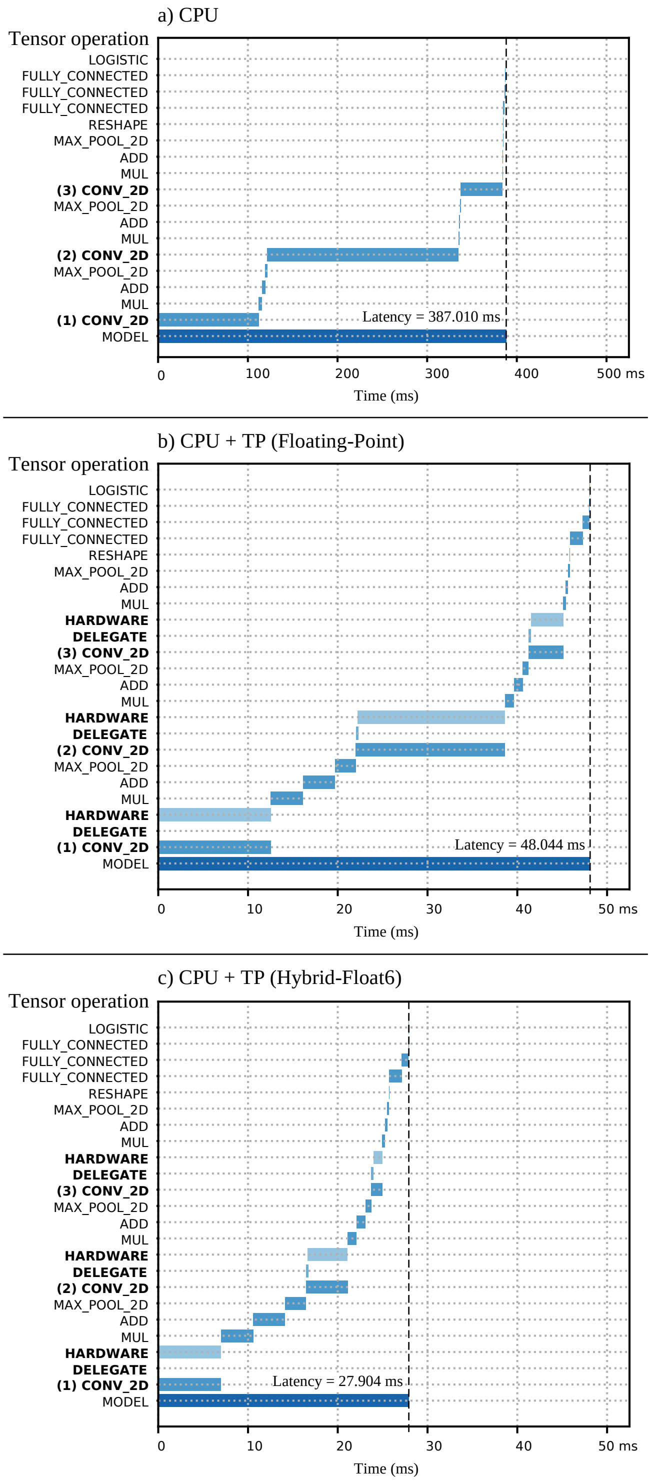

FIGURE 14. Inference run-time of TensorFlow Lite on the embedded system. (a) CPU ARM Cortex-A9 at 666MHz, (b) cooperative CPU + TP with floating-point Xilinx LogiCORE IP at 200MHz, and (c) cooperative CPU + TP with Hybrid-Float6 at 200MHz.

F. 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,

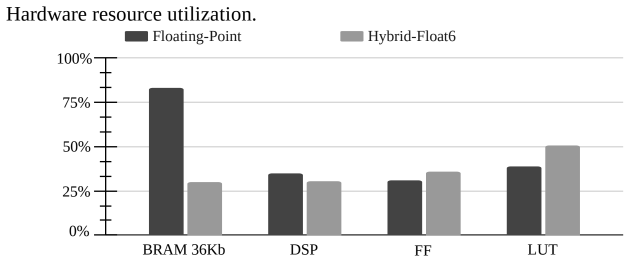

FIGURE 15. Hardware resource utilization on the Zynq-7007S SoC.

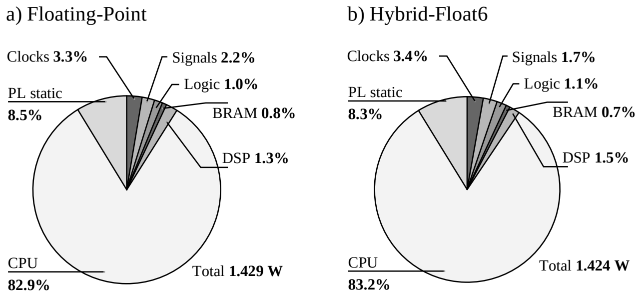

FIGURE 16. Estimated power dissipation on the Zynq-7007S SoC with PS at 666MHz and PL at 200MHz.

the inference on TP with standard floating-point presents a latency of 34.5ms, as result we get a $10.7\times$ 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 $55.5\times$ 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.** 1). 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. ??**. The platform implementations are summarized in **Tab. ??**, and their power dissipation breakdowns are presented in **Fig. ??**.

G. HARDWARE DESIGN EXPLORATION

To evaluate the methodology, we employ **Eq.** (7), 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. ??**, 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.** (3), 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.** (3), 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 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] 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.
[2] 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.
[3] 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.
[4] 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.
[5] 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.
[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] S. Sikdar, S. Banerjee, and G. Ashish, "Ultrasonic guided wave propagation and disbond identification in a honeycomb composite sandwich structure using bonded piezoelectric wafer transducers," *Journal of Intelligent Material Systems and Structures*, vol. 27, 10 2015.

[27] U. Kiencke, M. Schwarz, and T. Weickert, *Signalverarbeitung: Zeit-Frequenz-Analysen und Schätzverfahren*. Oldenbourh, 2008.

[28] R. B. Blackman and J. W. Tukey, "The measurement of power spectra from the point of view of communications engineering — part i," *Bell System Technical Journal*, vol. 37, no. 1, pp. 185–282, 1958.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

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

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