

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 Tecnología - 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 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 quantization, and compression storage [13], [14]; the second type of research corresponds to a more efficient data repre-

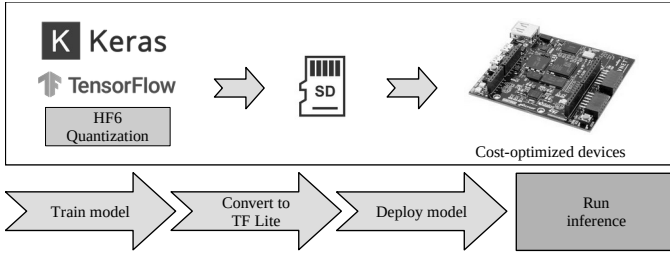


FIGURE 1. The workflow of our approach on embedded FPGAs.

sensation, 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 re-training, 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

In the literature we find plenty of hardware architectures dedicated to CNN accelerators implemented in FPGA and ASIC. Most of the research implements fixed-point quantization, and very limited research focuses on floating-point. Moreover, to the best of our knowledge, there is no research work focusing on floating-point inference for low-power embedded applications.

A. HARDWARE ARCHITECTURES TARGETING LOW-POWER AND COST-OPTIMIZED APPLICATIONS

Two research papers have been reported hardware accelerators targeting XC7Z007S. This is 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 not designed for CNN inference. 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

In [22], Liangzhen Lai et al. proposed a mixed data representation with floating-point for weights and fixed-point for

activations. In [23], Sean O. Settle et al. presented an 8-bit floating-point quantization scheme, which needs an extra inference batch to compensate for quantization error. However, [22] and [23] did not present a hardware architecture for their approaches. In [16], Xiaocong Lian et al. proposed an accelerator with optimized block floating-point (BFP), in this design the activations and weights are represented by 16-bit and 8-bit floating-point formats, respectively. This design is demonstrated on Xilinx VC709 PCIe evaluation board. This implementation achieves throughput and power efficiency of 760.83 GOP/s and 82.88 GOP/s/W, respectively.

C. LOW-PRECISION FLOATING-POINT

In [12], Chen Wu et al. proposed a low-precision (8-bit) floating-point (LPFP) quantization method for FPGA-based acceleration. This design is demonstrated on Xilinx Kintex 7 and Ultrascale/Ultrascale+. This implementation achieves throughput and power efficiency of 1086.8 GOP/s and 115.4 GOP/s/W, respectively.

III. BACKGROUND

A. FLOATING-POINT NUMBER REPRESENTATION

Representation of every numeric value, in any number system, is composed of an integer and a fractional part. The boundary that delimits them is called the radix point. The fixed-point format for representing numeric values derives its name from the fact that in this format, the radix point is fixed in a certain position. For integers this position is immediately to the right of the least significant digit.

In scientific computation, it is frequently necessary to represent very large and very small values. This is difficult to achieve using the fixed-point format because the bitwidth required to maintain both the desired precision and the desired range grows large. In such situations, floating point formats are used to represent real numbers. Floating point formats resemble scientific notation, such as -3.2004×10^{17} . Every floating point number can be divided into three fields: sign s , exponent e , and fraction f . Using the binary number system, it is possible to represent any floating-point number as:

$$(-1)^s \times 1.f \times 2^{e-BIAS} \quad (1)$$

Note that the exponent is biased, meaning that the stored value is shifted from 0 by a given value that depends on the bitwidth of the exponent field in the particular format. Also, the fraction represents the portion of the mantissa to the right of the radix point, while the term mantissa refers to the fractional and integer part.

A natural tradeoff exists between smaller bitwidths requiring fewer hardware resources and higher bitwidths providing better precision. Also, within a given total bitwidth, it is possible to assign various combinations of bitwidths to the exponent and fraction fields, where wider exponents result in higher range and wider fractions result in better precision.

The most widely used format for floating point arithmetic is the IEEE standard 754 [1]. This standard details four

floating point formats - basic and extended, each in single and double precision bitwidths. The IEEE single precision format is the same as shown in Equation 1 with $BIAS = 127$, 8 bits for the exponent and 23 bits for the fraction, or a total of 32 bits. In IEEE format, numbers are normalized, and only the fractional part is stored.

Optimal implementations of algorithms frequently do not require the bitwidths specified by the IEEE standard. Often, much smaller bitwidths than those specified in the 754 standard are sufficient to provide the desired precision. Reduced bitwidth implementations require fewer resources and thus allow for more parallel implementations than using the full IEEE standard format. In custom hardware designs, it is possible, and indeed desirable, to have full control and flexibility over the exact floating point format implemented. Our library provides this flexibility.

B. CONV2D TENSOR OPERATION

The *Conv2D* 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 [24]. We denote *Conv* as *Conv2D* operator.

$$Conv(W, h)_{i,j,o} = \sum_{k,l,m}^{K,L,M} h_{(i+k,j+l,m)} W_{(o,k,l,m)} + b_o \quad (2)$$

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.

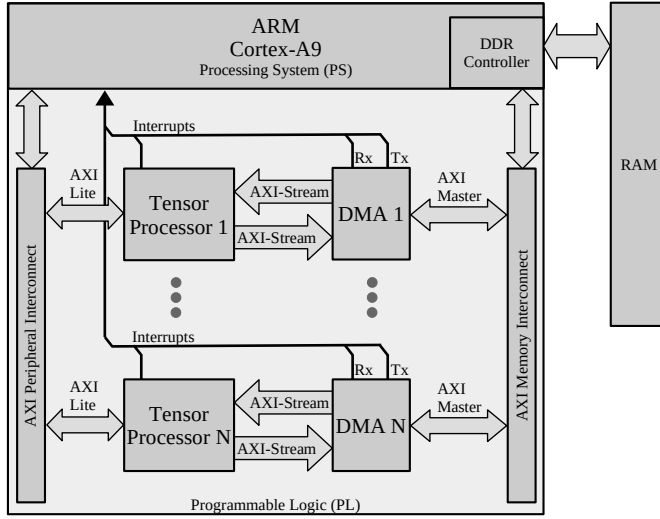


FIGURE 2. Base embedded system architecture.

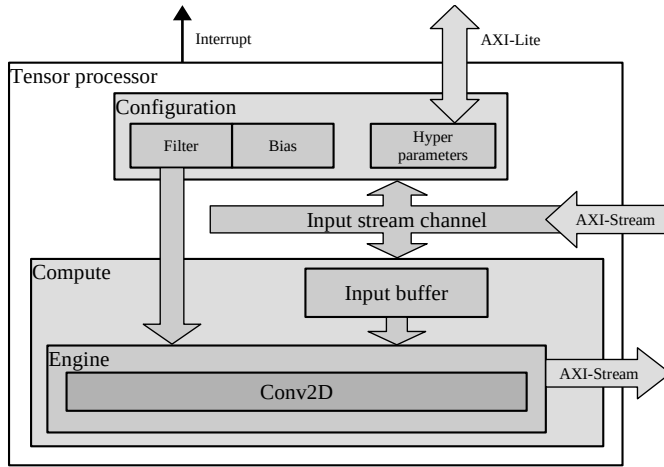


FIGURE 3. Hardware architecture of the proposed tensor processor.

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.

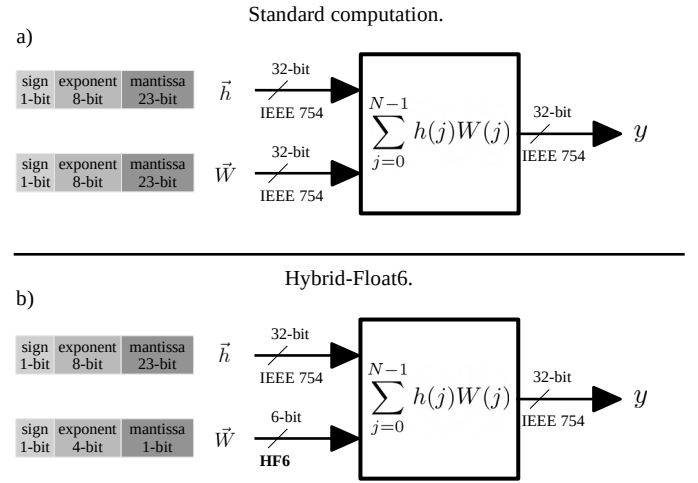


FIGURE 4. Dot-product hardware module with (a) standard floating-point and (b) Hybrid-Float6.

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. (3), 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.

$$L_{hf} = N + 7 \quad (3)$$

3) On-chip memory utilization

The total on-chip memory utilization on the TP is defined by Eq. (4), where TP_B and V_M represent the tensor buffers and local variables required for the design, respectively. Eq. (5) defines the tensor buffers, where $Input_M$ is the *input buffer*, $Filter_M$ is the *filter buffer*, $Bias_M$ is the *bias buffer*. The on-chip memory buffers are defined in bits. Fig. 6 illustrates the convolution operation utilizing the on-chip memory buffers.

$$TP_M = TP_B + V_M \quad (4)$$

$$TP_B = Input_M + Filter_M + Bias_M \quad (5)$$

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

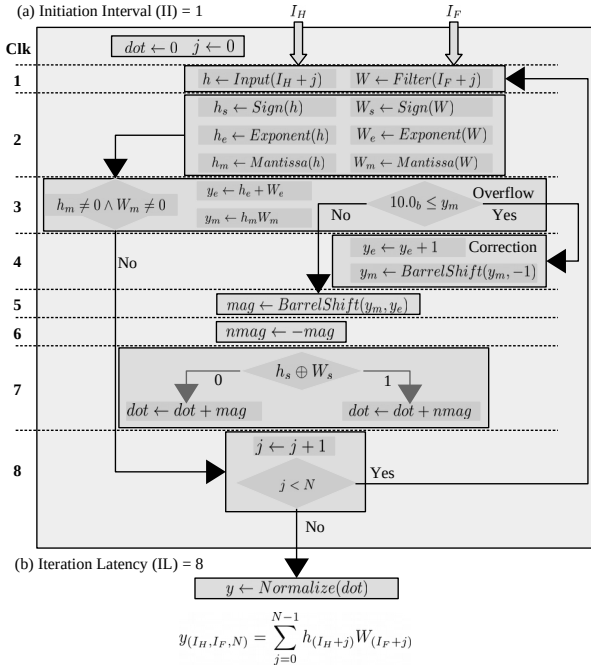


FIGURE 5. Pipelined hardware module for vector dot-product with hybrid custom floating-point, (a) exhibits the initiation interval of 1 clock cycle, and (b) presents the iteration latency of 8 clock cycles. I_H and I_F represent the input and filter buffer indexes, respectively.

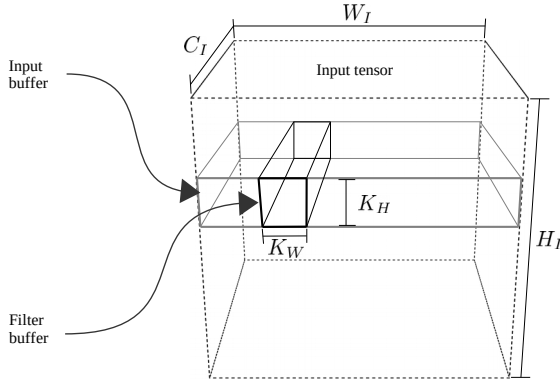


FIGURE 6. Design parameters for on-chip memory buffers on the TP.

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

$$Input_M = K_H W_I C_I BitSize_I \quad (6)$$

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.

$$Filter_M = C_I K_W K_H C_O BitSize_F \quad (7)$$

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

is the bit size of each bias element.

$$Bias_M = C_O BitSize_B \quad (8)$$

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.

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

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 aware training (QAT) 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.

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**
 $x \leftarrow (-1)^{sign} \cdot 2^{fullexp} \cdot (1 + (1 - 2^{-M_{size}}))$
 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

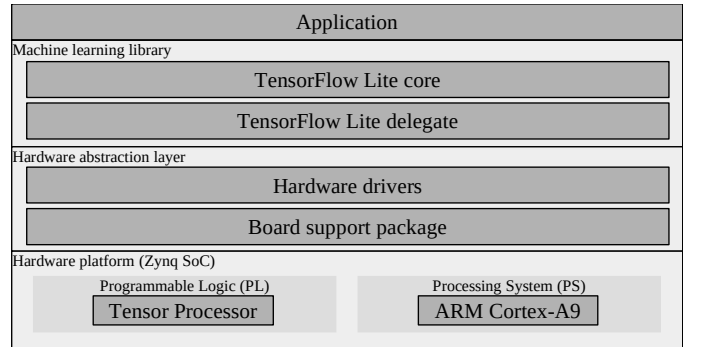


FIGURE 7. Base embedded software architecture.

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 32-bit, fixed-point 8-bit, Hybrid-Logarithmic 6-bit, and Hybrid-Float6.

To demonstrate the 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 Hybrid-Float6 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. In this subsection, we present the structure for 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). This illustrates a grid layout used to collect samples for the data sets. This grid is 10 × 10 and it does not consider the four corners as they are used for structure holders.

In order to create reproducible acoustic emissions, we use 9-cycle sine pulse in a Hanning window with central

frequency f_c (narrow-banded in the frequency domain). 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:

$$x_{\text{pulse}}(t) = \frac{1}{2} \left(1 - \cos \frac{f_c t}{5} \right) A_0 \sin f_c t. \quad (10)$$

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.

The signals for labeled pulses and noise disturbance 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 $f_s = 10 \text{ MHz}$. The disturbance signal is gaussian noise 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).

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). This is calculated as follows [27]:

$$\mathcal{F}_{m,k}^\gamma = \sum_{n=0}^{N-1} x[n] \cdot \gamma^*[n - m\Delta M] \cdot e^{-\frac{j2\pi kn}{N}} \quad (11)$$

Here $x[n]$ describes a discrete-time signal and $\gamma^*[n - m\Delta M] \cdot e^{-\frac{j2\pi kn}{N}}$ 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, $m = 0, 1, \dots, M - 1$ is valid. This complex-valued STFT is converted to real numbers via the magnitude square for pictorial representation in a spectrogram $\mathcal{S}_{m,k}$:

$$\mathcal{S}_{m,k} = \left| \mathcal{F}_{m,k}^\gamma \right|^2 = \left| \sum_{n=0}^{N-1} x[n] \cdot \gamma^*[n - m\Delta M] \cdot e^{-\frac{j2\pi kn}{N}} \right|^2 \quad (12)$$

In addition, these spectrograms are scaled in decibels. The spectrogram in decibels $\mathcal{S}_{m,k,\text{dB}}$ results in $\mathcal{S}_{m,k,\text{dB}} = 20 \cdot \log_{10}(\mathcal{S}_{m,k})$. For the conversion of the data, we use a signal length of 400 μs (75 μs pretrigger and 325 μs post trigger). Thus, the arrival times of the pulses are included in the spectrogram for all channels and labeled positions. We use a Blackman window function [28], a Fast Fourier Transform (FFT) length of 32 samples, and an overlap of 8 samples. 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).

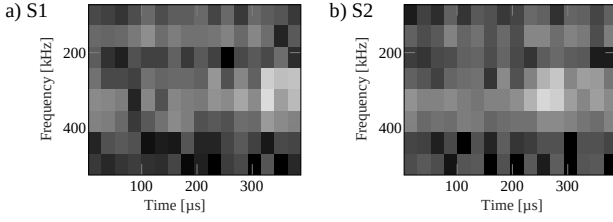


FIGURE 8. Spectrograms of sensors S_1, S_2 converted to grayscale for pulses at $x = 0.105, y = 0.109$ with noise disturbance.

In order to generate larger data sets, we create four further variants with time shifts of 15 μs / 30 μs / 45 μs / 60 μs . Subsequently, all spectrograms are converted to grayscale with scaling between -100dB and -40dB, see **Fig. 8**. Overall, the data set has a size of 500 (pulses) \cdot 5 (spectrograms) \cdot 6 (listening sensors) \cdot 96 (excitation points) = 1,440,000 images.

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:

- Input tensor.** This is 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)**.
- 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 .
- 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 $\epsilon = 1\text{e-}8$. The training cycle patience is 10 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 gets updated when the optimizer reaches a better minimum.

The resulting 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 53 minutes using

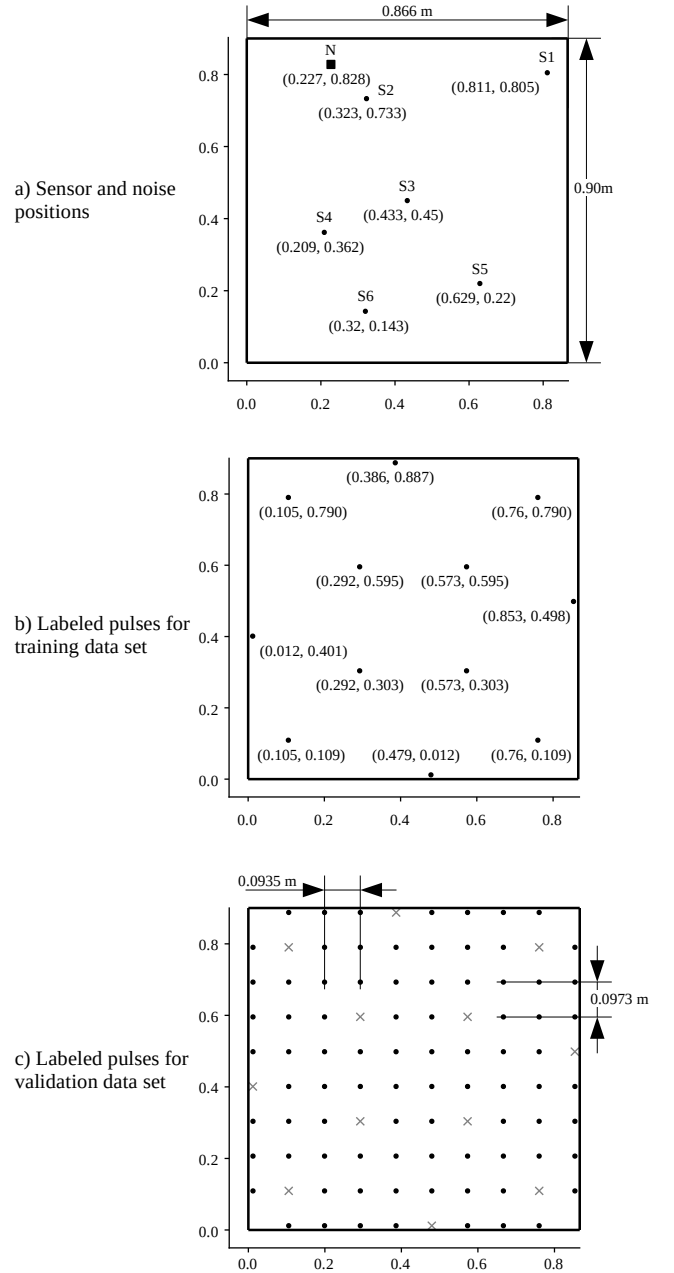


FIGURE 9. Experimental setup for sensor analytics on structural health monitoring.

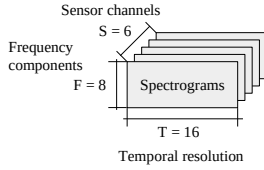
a PC with AMD Ryzen 5 5600H and NVIDIA GeForce RTX 3050.

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

a) Input tensor



CNN-regression model

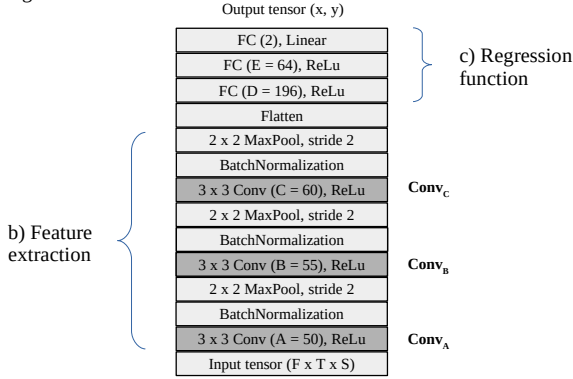


FIGURE 10. CNN-regression model for sensor analytics.

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 fixed-point model achieves $MSE = 0.0122m^2$ and $MAE = 0.0952m$. See Fig. 12(b). The MAE obtains a reduction of 0.31% compared to the base model. We attribute this to the regularization effect.

3) Quantization Aware Training for Hybrid-Float6

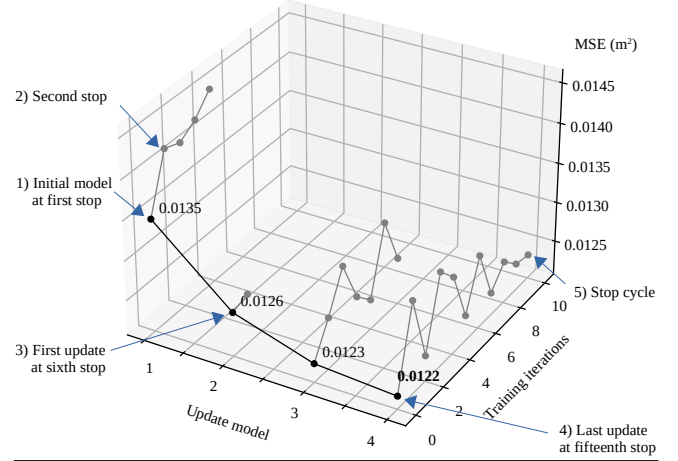
The QAT is a post-training step, this is applied with the method described in Algorithm 2. We run this method during two epochs with mini-batch size of 10 samples with 4-bit exponent and 1-bit mantissa as parameters. ($N_{ep} = 2$, $B_{size} = 10$, $E_{size} = 4$, $M_{size} = 1$).

The QAT is illustrated in Fig. 11(b). First, the model gets quantized with HF6 format (before QAT), this obtains $MSE = 0.0188m^2$ and $MAE = 0.1232m$. This illustrates the inference of the base floating-point model on HF6 hardware. See Fig. 12(c). The final model after QAT achieves $MSE = 0.0112$ and $MAE = 0.0919$. This is an error reduction of 8.2% and 3.77%, respectively. See Fig. 12(d). The QAT time is 185 minutes.

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

For the sake of comparison, we generate the model with 6-bit logarithmic quantization on convolution parameters. This quantization matches the hardware resource utilization of HF6. We run the QAT on the base model with hybrid loga-

a) Training with iterative early stop.



b) Quantization aware training

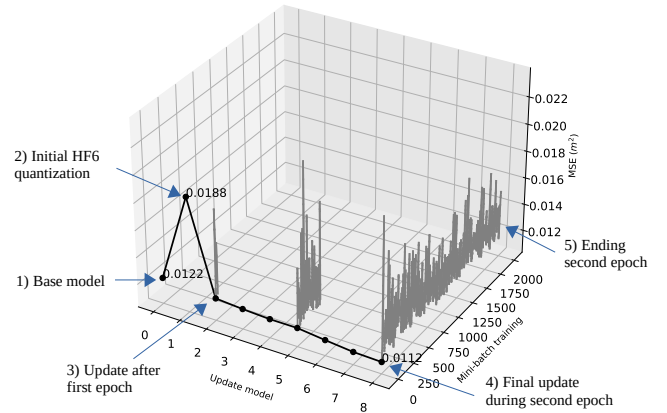


FIGURE 11. Training results.

arithmic parameters. The filter and bias tensors of convolution layers 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 reaches $MSE = 0.0123$ and $MAE = 0.0968$, which correspond to an error increase of 0.82% and 1.36%, respectively. See Fig. 12(e).

A summary of the inference with different data representations is presented in Fig. 12(f).

C. HARDWARE DESIGN EXPLORATION

The proposed hardware/software co-design is demonstrated on the Zynq-7007S system-on-chip (SoC) on the MiniZed development board. This SoC 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.

In this platform, we implement the proposed hard-

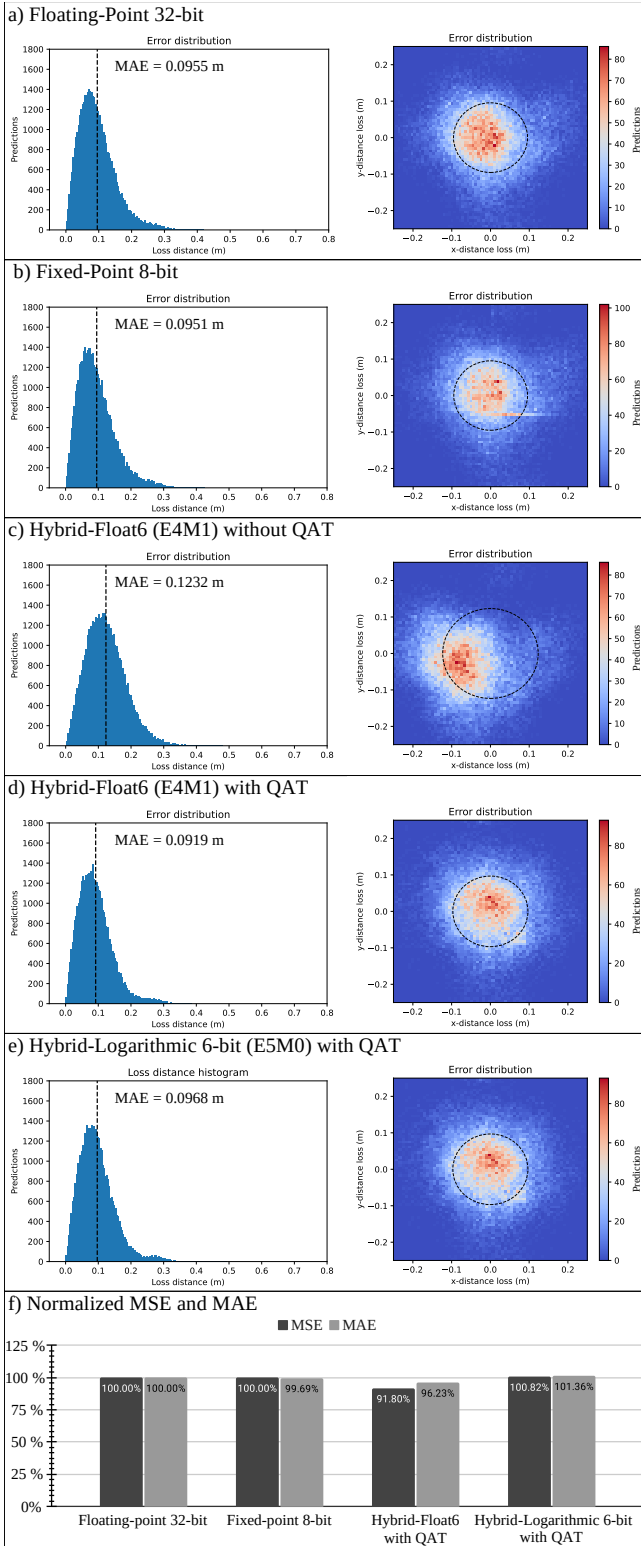


FIGURE 12. Performance of the model with different data representations.

ware/software architecture to deploy the sensor analytics application. The desired model is converted to TensorFlow Lite (floating-point) and deployed on the SoC as a hex dump

in a C array, this is used for the embedded software build. The Zynq-7007S SoC performs inference running TensorFlow Lite on the PS. The computational workload of convolution layers is delegated to the dedicated hardware.

1) Benchmark on Embedded CPU

We examined the performance of the embedded CPU for inference without hardware acceleration. In this case, TensorFlow Lite creates the CNN as a sequential model that allocates all computation to the CPU (ARM Cortex-A9) at 666 MHz and power dissipation of 1,187W.

The compute performance and run-time inference of the CPU are shown in **Tab. 2(a)** and **Fig. 14(a)**, respectively.

2) Benchmark on Tensor Processor with Standard Floating-Point Hardware using Xilinx LogiCORE IP

For this design, we implement the TP with standard floating-point hardware. The design parameters are:

- Max convolution kernel size: $K_W = K_H = 3$.
- Max input tensor width: $W_I = 16$.
- Max input and output channels: $C_I = 55$, $C_O = 60$.
- Filter and bias bit-size: $BitSize_F = BitSize_B = 32$.
- Input tensor bit-size: $BitSize_I = 32$.

Using equations from Section IV-B3, the on-chip memory buffer utilization are $Input_M = 84,480b$, $Filter_M = 950,400b$, and $Bias_M = 1,920b$. Hence, the required on-chip memory buffer size is $TP_B = 1,036,800b$.

The post-implementation resource utilization and power dissipation are presented in **Tab. 1(a)**. The complete hardware platform utilizes 83% of BRAM, this includes the on-chip memory requirements of the TP, DMA, and AXI interconnects. The total available on-chip memory (BRAM) on the Zynq-7007S SoC is 1.8Mb. The estimated power dissipation of the TP is 85mW at 200MHz (this estimation is provided by Xilinx Vivado).

TABLE 1. Resource utilization and power dissipation on the Zynq-7007S SoC.

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

The compute performance and inference schedule of the model on this hardware implementation is shown in **Tab. 2(b)** and **Fig. 14(b)**, respectively. In this implementation, TensorFlow Lite delegates computation of $Conv2D$ tensor operations to the TP.

The implementation of dot-product with standard floating-point arithmetic (IEEE 754) utilizes proprietary multiplier and adder floating-point operator cores. Vivado HLS implements 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

TABLE 2. Compute performance of the CPU and TP on each Conv2D tensor operation. This table presents: tensor operation, computational cost, latency, throughput, power efficiency, and estimated energy consumption.

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

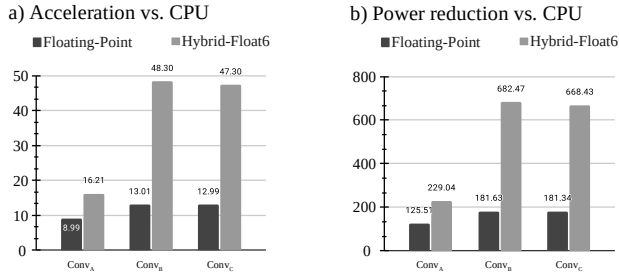


FIGURE 13. Inference acceleration and power reduction on the TP with floating-point and HF6 vs. CPU on the Zynq-7007S SoC.

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. 3**.

TABLE 3. Resource utilization and power dissipation of multiplier and adder floating-point (IEEE 754) operator cores (Xilinx LogiCORE IP).

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

3) Tensor Processor with Hybrid-Float6 Hardware

For the proposed design, the TP with HF6 hardware restates the standard floating-point design with the following customization: $BitSize_F = BitSize_B = 6$.

Using equations from Section IV-B3, the on-chip memory requirements are $Input_M = 84,480b$, $Filter_M = 178,200b$, $Bias_M = 360b$. Hence, the required on-chip memory buffer size is $TP_B = 263,040b$.

The post-implementation resource utilization and power dissipation are presented in **Tab. 1(b)**. The complete hardware platform utilizes 30% of BRAM, this includes the on-chip memory requirements of the TP, DMA, and AXI interconnects. The estimated power dissipation of the TP is

$84mW$ at $200MHz$ (this estimation is provided by Xilinx Vivado).

The compute performance and inference schedule of the model on this hardware implementation is shown in **Tab. 2(c)** and **Fig. 14(c)**, respectively. **Fig. 13** presents a comparison of acceleration and reduction of power dissipation between floating-point and HF6 hardware implementations.

D. DISCUSSION

1) Training and Quantization

The training with iterative early stop obtains a model with enhanced accuracy than using plain early stop. This method iteratively resets the moving averages of Adam's optimizer, which helps to search for better local minima. This iterative search is suitable for models with low computational cost; for example, low power sensor data analytics.

The TensorFlow Lite 8-bit quantization preserves overall accuracy. The regularization effect can improve the solution. However, the error distribution in CNN linear regressions gets slightly degraded. **Fig. 15(b)** shows this effect on three different models, where vertical and horizontal patterns appear in the error distribution of fixed-point quantization. We attribute this effect to the 8-bit resolution in the output activation maps. In the case of HF6 quantization, the activation maps are floating-point preventing this degradation.

The proposed HF6 quantization reduces power dissipation and preserves model accuracy. The 6-bit floating-point representation (E4M1) in the convolution parameters produces a better result than the 6-bit logarithmic representation (E5M0). This is demonstrated in **Fig. 12(d)** and **Fig. 12(e)**.

2) Hardware/Software Performance

The proposed HF6 hardware design reduces on-chip memory and DSP utilization while slightly increasing FF and LUT compared to the floating-point hardware implementation. See **Tab. 1** and **Fig. 16**. The HF6 hardware is implemented using FF and LUT, while the floating-point hardware is implemented using Xilinx LogiCORE IPs with DSPs.

The compute performance of the CPU and TP on each convolution layer of the model is presented in **Tab. 2** and **Fig. 13**. The peak acceleration and power efficiency of the TP with HF6 is $48.3\times$ and $5,743.29$ MFLOPS/s/W, respectively. The HF6 hardware demonstrates an improvement of $3.7\times$ in acceleration and power efficiency with respect to the standard floating-point hardware. See **Fig. 13**. The estimated power dissipation on the SoC is presented in **Fig. 17**. This shows a very similar breakdown of power dissipation in both implementations. However, the energy efficiency is increased due to the higher speed in HF6 hardware.

The run-time inference of TensorFlow Lite on the SoC is illustrated in **Fig. 14**. This shows a cooperative performance where the convolution tensor operations are delegated to the dedicated hardware accelerator. The platform with floating-point hardware accelerator obtains a latency of $48ms$, while the implementation with HF6 obtains a latency of $27.9ms$.

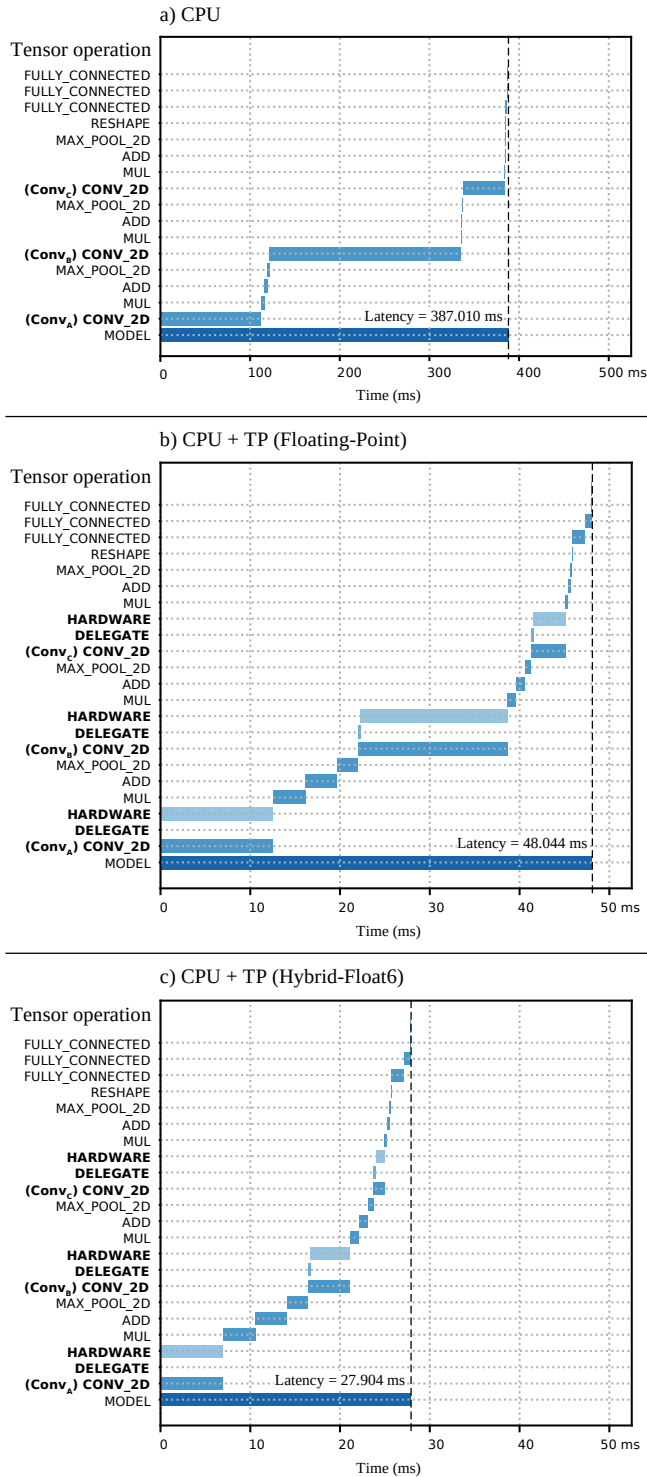


FIGURE 14. Run-time inference of TensorFlow Lite on the Zynq-7007S SoC. (a) CPU ARM Cortex-A9 at 666MHz, (b) cooperative CPU + TP with floating-point Xilinx LogiCORE IP at 200MHz, and (c) cooperative CPU + TP with Hybrid-Float6 at 200MHz.

These represent an overall acceleration of $8\times$ and $13.87\times$, respectively.

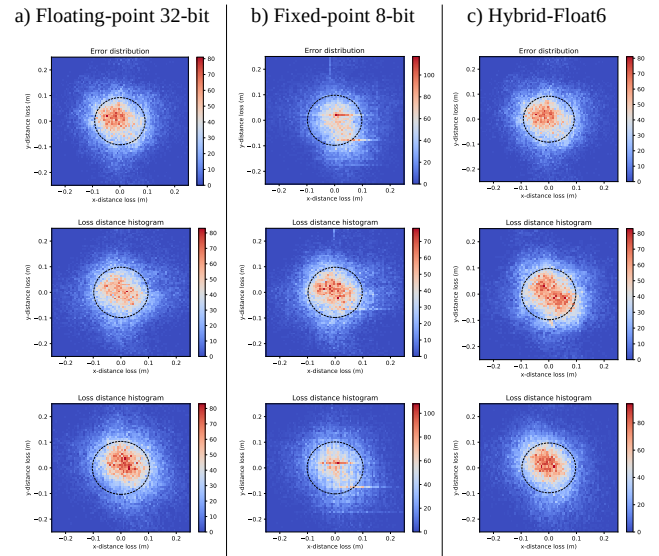


FIGURE 15. 2D error distribution of three CNN-regression models.

Hardware resource utilization.

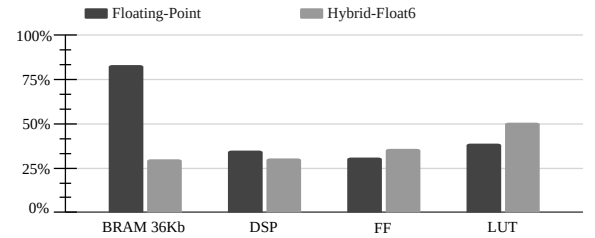


FIGURE 16. Hardware resource utilization on the Zynq-7007S SoC.

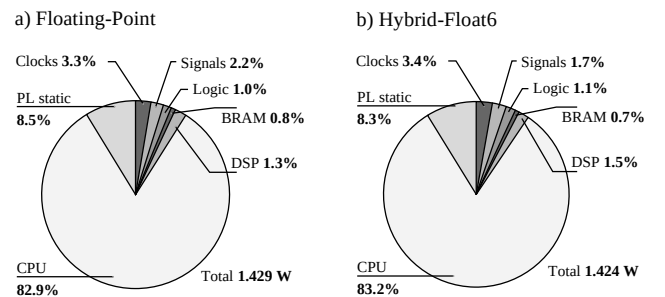


FIGURE 17. Estimated power dissipation on the Zynq-7007S SoC with PS at 666MHz and PL at 200MHz.

3) SoC Design and Compatibility

The proposed design is an alternative for low-power floating-point ML inference with a cooperative hardware/software mechanism. This architecture delegates compute-bound tensor operations to a dedicated hardware accelerator, while memory-bound operations are processed by the CPU.

The 32-bit floating-point representation in activation maps enables backward ML compatibility. The 6-bit floating-point representation (E4M1) reduces the 24-bit mantissa multiplication to 1-bit flag operation. This reduces latency, hardware

area, and power consumption.

Backwards ML compatibility gives portability from training to inference. This enables to run inference of HF6 quantized models on standard floating-point hardware and vice versa. Running inference of models without QAT enables rapid deployment; however, this will incur in accuracy degradation, see Fig. 12(c).

4) Future Work

As future work, we plan to investigate further improvements in energy efficiency. To reduce energy consumption, activation maps can be represented with brain floating-point. This would reduce hardware resource utilization, memory footprint, and data transfer. To increase performance, this implementation would require matching computational throughput with memory bandwidth using parallelization approaches.

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 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) demonstrate peak acceleration and power efficiency of 48.3X and 5.7 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. Loccupier, 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. Avcı, 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, "EdgerNN: 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. Rotermond, 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*. Oldenbourg, 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.

...