IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 28 June 2022, accepted 5 July 2022, date of publication 11 July 2022, date of current version 18 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3189776

RESEARCH ARTICLE

Custom Hardware Inference Accelerator for TensorFlow Lite for Microcontrollers

EREZ MANOR[1] AND SHLOMO GREENBERG[1,2], (Member, IEEE)
[1]Department of Electrical and Computer Engineering, Ben-Gurion University, Beer-Sheva 84105, Israel
[2]Department of Computer Science, Sami Shamoon College of Engineering, Beer-Sheva 84100, Israel

Corresponding author: Shlomo Greenberg (shlomo.greenberg@gmail.com)

This work was supported in part by the High-Tech Scholarship Award, and in part by the GenPro Consortium within the framework of the Israel Innovation Authority's Magnet Program.

ABSTRACT In recent years, the need for the efficient deployment of Neural Networks (NN) on edge devices has been steadily increasing. However, the high computational demand required for Machine Learning (ML) inference on tiny microcontroller-based IoT devices avoids a direct software deployment on such resource-constrained edge devices. Therefore, various custom and application-specific NN hardware accelerators have been proposed to enable real-time Machine Learning (ML) inference on low-power and resource-limited edge devices. Efficient mapping of the computational load onto hardware and software resources is a key challenge for performance improvement while keeping low power and a low area footprint. High performance and yet low power embedded processors may be attained via the usage of hardware acceleration. This paper presents an efficient hardware-software framework to accelerate machine learning inference on edge devices using a modified TensorFlow Lite for Microcontroller (TFLM) model running on a Microcontroller (MCU) and a dedicated Neural Processing Unit (NPU) custom hardware accelerator, referred to as MCU-NPU. The proposed framework supports weight compression of pruned quantized NN models and exploits the pruned model sparsity to reduce computational complexity further. The proposed methodology has been evaluated by employing the MCU-NPU acceleration for various TFLM-based NN architectures using the common MLPerf Tiny benchmark. Experimental results demonstrate a significant speedup of up to 724x compared to a pure software implementation. For example, the resulting runtime for the CIFAR-10 classification is reduced from about 20 sec to only 37 ms using the proposed hardware acceleration. Moreover, the proposed hardware accelerator outperforms all the reference models optimized for edge devices in terms of inference runtime.

INDEX TERMS TinyML, neural processing unit, TensorFlow-Lite for microcontrollers, hardware-software codesign.

I. INTRODUCTION

In recent years, the need to deploy artificial neural networks and machine learning algorithms on embedded platforms, such as Internet-of-Things (IoT) edge devices has been steadily increasing. Running deep learning models on these IoT devices, enables data analytics to be directly performed at the edge near the sensor, expanding the scope of Artificial Intelligence (AI) applications [1].

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

The IoT devices are based on low-cost, low-energy microcontrollers (MCUs). Although Machine learning (ML) inference is one of the most required applications on these IoT devices, the high computational demand for ML inference avoids a direct real-time deployment on such resource-constrained edge devices. Applying ML inference at the edge, particularly using low-power MCUs, is gaining increased interest in the industrial and academic ML community.

Custom hardware accelerators and application-specific Neural Networks (NN) hardware accelerators enable real-time ML inference on low-power and resource-limited edge devices [1]. Various novel quantization and compression methods, as well as new training approaches, have been developed to reduce the computational cost and memory footprint of NN to fit the limited computing capability and storage capacity of MCU-class devices [2], [3].

TinyML is a unique approach to edge computing exploring machine learning models to be efficiently deployed and trained on MCU edge devices [1]. The TinyML enables running ML models on edge devices with low-power and low-memory microcontrollers by integrating quantization, compression, and optimized machine learning techniques.

To facilitate the deployment of TinyML models on MCUs, several specific software frameworks and integrated NN libraries have been developed. Robert *et al.* [4] introduce an open-source ML inference framework, so called the TensorFlow Lite Micro (TFLM), for running deep-learning models on limited-resources embedded systems. TFLM is characterized by low resource requirements and minimal runtime performance overhead and, therefore, can tackle the embedded-system resource constraints such as limited memory and limited computation power.

A hardware-software efficient codesign is required to fully utilize the potential of efficient ML inference in MCU edge devices [5]. Using a general-purpose microcontroller for such systems typically results in a design that fails to meet the NN application-specific requirement [6]. To achieve the desired requirements, one may use extensible MCUs, which offer the flexibility of adding custom ML hardware accelerators and enabling real-time NN processing on small-scale edge devices. Various FPGA-based hardware platforms provide the ability to add custom hardware accelerators to common microcontrollers [7], [8].

A new generation of micro-controller-based Neural Processing Units (micro NPUs) promises efficient ML inference and low-power embedded execution of AI workloads while supporting compression and pruning [9]. This work proposes an efficient NPU to accelerate ML inference on edge devices using a modified TFLM model and a unique custom hardware accelerator. The proposed framework supports weight compression of pruned quantized NN models and exploits the pruned model sparsity for reducing the computational complexity.

A TFLite model is represented in a format known as a Flatbuffer, which includes all the information regarding a given network model. An efficient representation of the network model that is adapted to a specific hardware implementation is proposed. The TFLM Flat buffer is reorganized to separate the information regarding the weights matrices from the network model. This results in a significant reduction of the FlatBuffer size and enables storing the compressed weights in external memory.

The key contributions of this paper are as follows:

- An efficient hardware-software codesign for hardware acceleration of TFLM inference on edge devices.
- A new framework for efficient and compact representation of the TFLM network model separating the weights from the network model and enabling a low-memory footprint for applying the network model.
- Adding real-time decompression capabilities to the common TFLM framework by integrating HW-based lossless DNN weight compression approach enabling on-the-fly decoding of one weight per clock cycle.
- Leveraging fine-grained pruning (yielding high sparsity) for TFLM network models and using RLE to efficiently represent a sequence of zero weights enabling significantly faster computation.
- Evaluating the proposed method using the common MLPerf Tiny benchmark, demonstrating a significant speedup compared to a pure software implementation.

The rest of this paper is organized as follows: Section II presents an overview of related work. Section III provides a thorough description of the proposed approach. Finally, experimental results are presented in Section IV, while conclusions are given in Section V.

II. RELATED WORK

This section reviews some related work in the area of AI inference on edge devices. First, a review of common existing frameworks and available deep learning compiler toolchains are presented. Then, the focus is on some optimized TFLM implementation for specific hardware and custom hardware-based models for neural networks.

A. EMBEDDED AI SOFTWARE FRAMEWORKS

Recently, several software frameworks and integrated libraries have been developed to facilitate the deployment of ML models on microcontrollers and edge devices.

Robert *et al.* [4] introduce the TensorFlow Lite Micro (TFLM), an open-source ML inference framework for running deep-learning models on embedded systems. TFLM tackles the embedded-system resource constraints such as limited memory and limited computation power. TFLM is characterized by low resource requirements and minimal runtime performance overhead. To optimize the required memory size and improve latency, TFLM applies both quantization and weight pruning.

To enable efficient AI inference on edge devices, STMicroelectronic present a unique dedicated framework, the X-Cube-AI for the STM32 microcontroller [10]. The X-Cube-AI software tool converts pre-trained neural networks into memory and computation optimized C library. The tool also offers the possibility to compress the model to decrease the memory size with minimal accuracy loss. Quantization is used to convert weights and activations from 32-bit floating-point to 8-bit integer precision.

Several works [11]–[13] present a benchmark comparison between the two popular frameworks. Although the X-Cube-AI seems to outperform the TFLM in terms of average inference time and memory size, it is a proprietary library that supports only the STM32 device, while the TFLM is open-source, widely available, and can be applied to various MCUs.

Microsoft has released an open-source library Embedded Learning (ELL) [14], suggesting a framework that enables the deployment of pre-trained ML models on constrained platforms. Facebook has developed a machine learning compiler, so-called Graph Lowering (GLOW) [15], that accelerates the performance of deep learning frameworks on different hardware platforms. Ji. L. *et al.* [16], [17] propose a unique framework, so called MCUNet, that enables efficient AI inference on low-power microcontrollers. This work focus on the common TFLM framework due to its easy and efficient deployment [11].

B. TFLM KERNEL OPTIMIZATION

This section reviews some optimized TFLM implementations for specific hardware.

An efficient NN kernel targeted for optimized AI inference on intelligent IoT edge devices, the CMSIS-NN, is presented by L. Lai *et al.* [18]. Arm proposes an optimized version of the TensorFlow Lite kernels that use CMSIS-NN to deliver fast performance on Arm Cortex-M cores. When running neural networks, CMSIS-NN reduces the cycle count by 78.3%, reducing energy consumption by 79.6%. The software library CMSIS-NN achieves these results using a Single Instruction/Multiple Data (SIMD) unit and quantizing the neural networks [19].

An open-source optimized library, the PULP-NN, which is based on the CMSIS-NN framework, is proposed by Garofalo *et al.* [2]. The PULP-NN includes a set of kernels and utilities to support an efficient inference of quantized NN on a DSP-optimized RISC-V-based processor [20]. By fully exploiting the DSP extensions available in the Instruction Set Architecture (ISA), they achieve a speedup factor of 9 with respect to the plain ISA. Optimization of the convolution kernel by improving the data reuse, yields a further 20% performance gain with respect to the original kernel of CMSIS-NN.

Jure V. *et al.* [19] propose a simple instruction set extension with two main components, hardware loops and dot product instructions. To evaluate the effectiveness of the extension, an optimized assembly functions for the fully connected and convolutional neural network layers have been developed. When using the extensions and the optimized assembly functions for CNN layers, they achieve an average clock cycle count decrease of 73% for a small-scale CNN network.

Custom hardware-based neural network accelerators can surpass general-purpose processors in terms of both throughput and energy efficiency [21]. Therefore, we propose to use dedicated hardware-based Neural Processing Units (NPUs) as the main accelerator engine.

C. NEURAL PROCESSING UNITS

A new generation of micro-controller-based Neural Processing Units (micro NPUs) promises efficient ML inference and low-power embedded execution of AI workloads while supporting compression and pruning [9].

Manor *et al.* [3] propose a hardware/software codesign partitioning methodology for ML inference acceleration using TFLM models. The proposed method is based on graph analysis of the NN model architecture and extracting common patterns into hardware accelerators, providing a speedup of up to 3 orders of magnitude compared to software-only implementation.

Leon H. *et al.* [22] present a unique development framework, providing optimized TFLite models and, on the other hand, hardware kernel generators for use with the proposed framework. The proposed flow automatically creates an edge AI computing platform with custom hardware accelerators designed specifically for the given NN model. Executing the model on the RISC-V NMU using Tensor-Flow Lite Micro yields a speedup factor of 48 for the fully-connected operations. A modified TFLM model replaces the model's accelerated operations with a custom operation, which is integrated into the TFLM runtime. However, the presented framework is limited in its ability to support generic models with different topologies.

A proprietary microNPU accelerator designed by Arm is aimed to accelerate neural network inference on Cortex-M with a low area and low power consumption [23]. The microNPU shares the neural network processing with the host Cortex-M. An offline optimizer generates a TFLite FlatBuffer file which is deployed on the target device. The FlatBuffer file contains information on either the specific neural network layer is executed on the microNPU or the Cortex-M processor. Paired with a Cortex-M processor, the microNPU delivers up to a 137x ML performance uplift compared to previous Cortex-M generations [24].

This work proposes an efficient NPU in terms of Power-Performance-Area (PPA) to accelerate ML inference on MCU using a modified TFLM model and custom hardware accelerator. The TFLM flat file is reorganized to contain only the NN model, while the weights matrices are extracted and located in external memory.

The proposed framework supports the weight compression of pruned NN models and exploits the pruned model sparsity to reduce the computational complexity. Both data compression and zero-skipping techniques relies on sparsity [25]. The sparsity measure represents the number of zero weights in the weight's matrix. In the pruning process a zero value is assigned to the weights which have low values (under a predefined thresholds), and therefore the sparsity is increased. The combination of the TensorFlow built-in pruning tool with the proposed compression algorithm [26] yield better results as the sparsity increases.

III. THE PROPOSED APPROACH

This section describes the proposed NPU and the accompanying framework in detail.

A. NPU ARCHITECTURE

Fig. 1 depicts the proposed NPU architecture and its interface to the MCU and external memory. The NPU performs

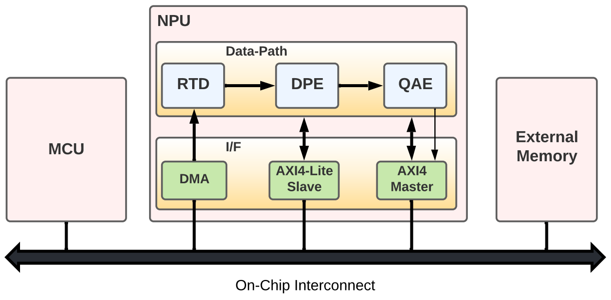

FIGURE 1. The proposed NPU architecture.

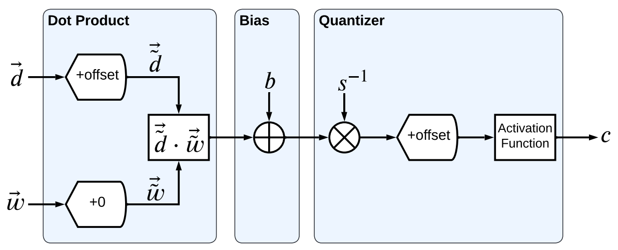

FIGURE 2. TFLite basic computation for a single neuron.

matrices multiplication of the weights and the data inputs and writes the activation data output to the external memory using the AXI4 master agent.

The proposed NPU is a loosely-coupled accelerator located outside the MCU. Both The MCU and the NPU can access shared memory through an on-chip interconnect (like the AXI4 bus). The NPU includes two main modules: the data-path and the interfaces (I/F). The data path is comprised of three main blocks: (a) a Real-Time Decoder (RTD), (b) a Dot Product Engine (DPE), and (c) a Quantization and Activation Engine (QAE). The I/F module contains three bus agents: (a) an AXI4-Lite slave, (b) a DMA master, and (c) an AXI4-master [27].

Each data-path element contains a register file (RF) array to store the configuration parameters received from the MCU. The MCU configures the RF through the use of the AXI4-Lite agent. The RTD block fetches and uncompresses the compressed weights from an external memory through the DMA agent. Then, the DPE module performs matrices multiplication of the weights and the data inputs. The QAE module is responsible for quantization, generating, and storing the activation data into the external shared memory. Both the DPE and QAE modules can access the shared memory using the AXI4 master agent. The QAE writes the activation data output to the external memory using the AXI4 master agent.

This work suggests adapting the TFLite 8-bit quantization scheme [28], to provide hardware support for inference with quantized TFLite models. The 8-bit quantization approximates floating-point values using Eq. 1, where the weights are represented as a signed integer with a zero-point equal to 0, and activations/inputs are represented as a signed integer with a zero-point in the range -128 to 127.

$$real\_value = (int8\_value - zero\_point) \times scale \tag{1}$$

The data path is designed in accordance with the TFLite convention and is a bit-accurate implementation of the TFLite software kernels. Fig. 2 depicts the basic computation required for every single neuron. Multiplying the input vector ($\vec{d}$) by the weights vector ($\vec{w}$), adding a bias (b), and quantization. The weights are of type int8 and the activations are of type uint8. First, the unit8 input vector is converted to a 32-bit integer (by adding a zero-point offset) and multiplied by the weights vector to generate 32-bit accumulating products. Then, a bias (b) is added and the result is scaled down (s^{-1}). To convert the result back to uint8 (for efficient storage) an offset is added. Finally, the activation function is applied to yield the unsigned 8-bit output.

1) REAL-TIME DECODER

This work use the HW-based real-time DNN lossless weight compression approach for the RTD module [26]. The compression algorithm is based on the Huffman algorithm and is applied to pre-partitioned weight classes according to the appearance probability of the weights. In the decompression phase, the encoded weights are extracted in a sequential manner, deploying weight-by-weight decoding. The RTD is used for fetching and decoding (i.e., decompressing) the compressed weights stored in the external memory.

The proposed RTD decompression hardware module is implemented using a four-stage pipeline, allowing the decoding of one weight per clock cycle [26]. The uncompressed weights are transferred to the DPE module through the AXI4 stream interface. For efficient compression, the weights matrices are first pruned using fine-grained pruning, which results in better sparsity. The TFLite also has support for pruning optimization. As a result, sparse matrices can be efficiently compressed [29]. H. Mao *et al.* show that in terms of accuracy, the fine-grained pruning (i.e., pruning of individual weights by applying a unique pruning criterion for each weight) gives the best results with up to 0.8 sparsity within the weights matrix [30].

2) DOT PRODUCT ENGINE

The Dot Product Engine (DPE) is responsible for performing the dot-product multiplication of the weights and data. Fig. 3 depicts the DPE architecture. The DPE module comprises the following four main components: a Register File (RF), Run-Length Encoding (RLE) block, Arithmetic Unit (AU), and a Control Unit (CU).

The MCU can configure the DPE mode of operation by setting the appropriate RF parameters through the AXI4-Lite. The following parameters are configurable: layer type (convolution or fully connected layer), input data address, input data offset, input data dimensions, filter data dimensions, output data dimension, and filter stride. The DPE is coupled with both data and weight memory.

The uncompressed weights are streamed from the RTD module to the DPE. Then, the DPE uses Run-length

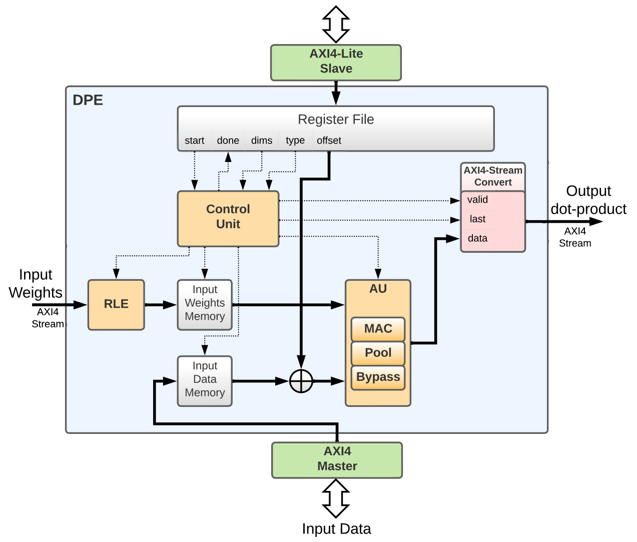

FIGURE 3. The proposed Dot Product Engine (DPE).

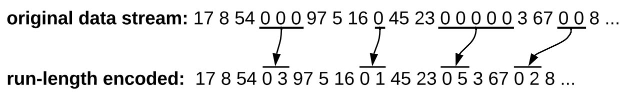
original data stream: 17 8 54 0 0 0 97 5 16 0 45 23 0 0 0 0 0 3 67 0 0 8 ...
run-length encoded: 17 8 54 0 3 97 5 16 0 1 45 23 0 5 3 67 0 2 8 ...

FIGURE 4. An example of the Run-Length Encoding (RLE) usage.

encoding (RLE) for efficient representation of a sequence of zero weights. Detecting those "zero" weights enables efficient and faster computation. Fig. 4 shows an example of the RLE encoder functionality. Sequences of "zero" are replaced by only one zero, followed by the number of the repeated "zeros".

The control unit is responsible for fetching the input data from the external memory and the weights from the RLE module. The input data are fetched through the AXI4 master bus and stored in internal memory. The control unit can perform a continuous partial data loading to reduce the internal memory size if required. This can be done without stalling the CPU, however, at a high cost of increased power consumption.

To keep the required memory for the weights small, only a relatively small set of weights are fetched from the AXI4 stream bus at a time for each dot product multiplication with the whole input data. Therefore, although a relatively small weight memory is required, the flow still allows a continuous and fast operation.

The CU allows using different data indexing types to support various NN layers. The proposed framework supports the following six layers: Convolution, Depthwise Convolution, Fully Connected, Max Pool, Avg Pool, and Softmax. The Arithmetic unit supports MAC operation, maximum and average pooling, and bypass operation (transferring the data directly to the QAE module for Softmax computation).

The last block is used as an AXI4-Stream protocol converter. For each value in the dot-product matrix (a scalar data) the CU adds a valid bit (meaning it can be used by the

Algorithm 1 Pseudocode of the Software Convolutional Layer

```
    Parameters: $U, V, F, R, C$
    Inputs: $X, W, B, S, O$
    Output: $Y$
 1: for $u = 0; u < U; u{+}{+}$ do
 2:     for $v = 0; v < V; v{+}{+}$ do
 3:         for $f = 0; f < F; f{+}{+}$ do
 4:             for $kh = 0; kh < R; kh{+}{+}$ do
 5:                 for $kw = 0; kw < R; kw{+}{+}$ do
 6:                     for $c = 0; c < C; c{+}{+}$ do
 7:                         $Y^f_{u,v} += (W^{f,c}_{kh,kw} + 0) \times (X^c_{u+kh,v+kw} + O_x)$;
 8:                     end for
 9:                 end for
10:             end for
11:             $Y^f_{u,v} += B^f$;
12:             $Y^f_{u,v} *= S^{f^{-1}}$;
13:             $Y^f_{u,v} += O_y{}^f$;
14:         end for
15:     end for
16: end for
```

QAE), and the last bit represents the end of the specific layer computation.

Algorithm 1 depicts the software implementation of the TFLM Conv2D layer. The Convolution process is carried out using the six nested loops (lines 1-6). The inputs X, W, B, S, and O stand for the input data matrix, weights matrix, bias, scaling vector, and the offsets vector, respectively. The output dot-product Y is calculated in line 7 and quantized in lines 11-13. The parameters U, V, and F represent the size of the 3D output matrix, while R and C define the size of the weight matrix.

Algorithm 2 depicts the pseudocode for the hardware implementation of the TFLM Conv2D layer. For an efficient hardware implementation, the order of the loops (used for filter selection) has been changed to enable applying the selected filter (W^f) to the whole input matrix (lines 1-3) in the same iteration. This results in reducing the number of the required memory access for fetching the filter bank from the external memory. The same hardware module can be utilized for both convolution and fully connected layers.

3) QUANTIZATION AND ACTIVATION ENGINE

The QAE module is responsible for data quantization, generating, and storing the activation data into the external shared memory (using the AXI4 master agent). The module comprises four main blocks: a register file (RF), a TFLite-based Scaler, an Activation Unit (AU), and a Control Unit (CU).

Fig. 5 depicts the QAE architecture. The AXI4 master bus interface includes two FIFOs: one is used to transfer the bias dot-product output and the other for the shift parameter derived from the TFLite. The quantization is carried out using

Algorithm 2 Pseudocode of the Hardware Convolutional Layer

```
    Parameters: $U, V, F, R, C$
    Inputs: $X, W, B, S, O$
    Output: $Y$
1: for $f = 0; f < F; f{+}{+}$ do
2:     load($W^f$);
3:     for $u = 0; u < U; u{+}{+}$ do
4:         for $v = 0; v < V; v{+}{+}$ do
5:             the same loops as in lines 4-13 of Alg. 1
6:         end for
7:     end for
8: end for
```

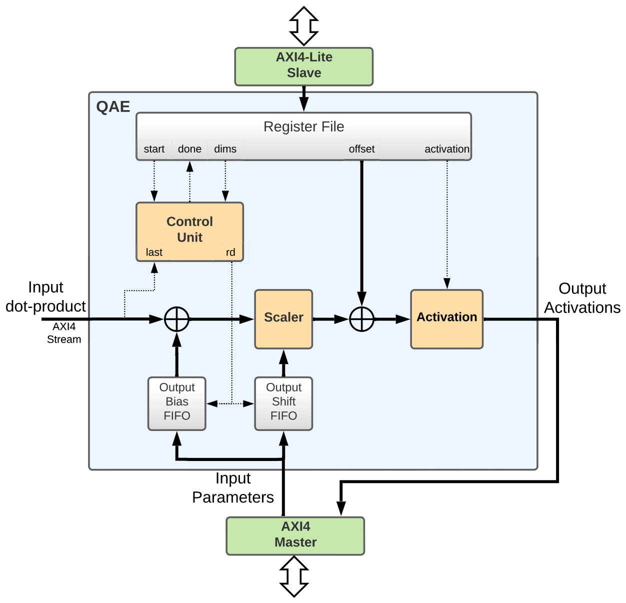

FIGURE 5. TFLite quantizer module (QAE).

the Scaler unit, which allows performing a set of conditional shift operations to convert the accumulated 32-bit into 8-bit value. Then, an offset (i.e., the zero-point in Eq. 1) is added, and finally, in the last phase, the Relu activation function is performed. The result is stored in the external memory via the AXI4 master.

B. FLATBUFFER CONVERSION

A TFLite model is represented in a format known as FlatBuffer (identified by the .tflite file extension). The FlatBuffer includes all the information regarding a given network model, i.e., the network topology, the various network parameters (such as input/output size, number of channels/filters, the activation type, etc.), weights matrices, and the configurable quantization parameters (offset and scaling).

A more efficient network model representation adapted to a specific hardware implementation is proposed. The main idea is to separate the information regarding the weights matrices from the network model. The new representation includes two different data structures: a mini-Flatbufer and a weights array. This results in a significant reduction of the FlatBuffer size (75% on average) and enables storing of the compressed weights in external memory.

Since the micro-controllers (used in edge devices) have limited internal memory storing the compressed weights in an external memory has a significant advantage enabling the implementation of a larger network model. The latency derived by accessing the external memory is hidden by using a prefetch mechanism using DMA and storing the prefetch weights in an internal relatively small buffer.

The proposed hardware accelerator utilizes the separation of the weights from the part representing the network model. This enables storing the whole network model in a relatively small internal memory (using cache or TCM) and saves the long latency derived from accessing an external memory. Moreover, the weights can be compressed and stored in external memory while the decoding is carried out on the fly in real-time, enabling the decoding of a single weight on each cycle using the proposed unique RTD module.

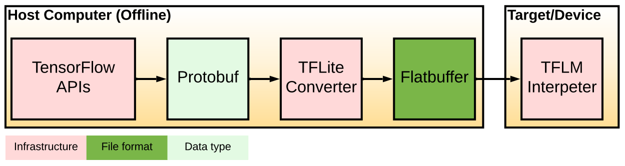

FIGURE 6. TensorFlow Lite inference flow.

Fig. 6 illustrates the standard flow for TFLite inference on a target MCU. The network is trained on a host computer using TensorFlow API to generate a network model which is stored in a unique structure called Protobuf. Then, the TFlite converter produces the FlatBuffer, which is an efficient serialized representation that keeps the memory footprint small and enables efficient inference implementation. A dedicated TFLM interpreter contains code to load and run the network models on the target device. The TFLM interpreter expects the model to be provided as a C++ array. Therefore, the FlatBuffer is converted into an appropriate C byte array. The TFLM library is compiled for the MCU and contains the interpreter, the FlatBuffer model, and information for the TFLite schema.

Fig. 7 depicts the proposed new flow. First, the original TFLite FlatBuffer is converted into a readable data structure file (.json). Then, the weights of each layer are extracted in order according to the network model topology. The extracted weights are stored in a single C array. The proposed mini-FlatBuffer represents the network model excluding the weights. To preserve the original program flow and enable access to the weights while using the mini-FlatBuffer, an index is provided for each layer representing the offset location of that layer in the extracted weights C-array. The TFLM library has been modified to support the proposed scheme and the new indexing used for accessing the appropriate weights.

Fig. 8 depicts the proposed hardware-software TFLM flow that is applied to the target device. The target device is

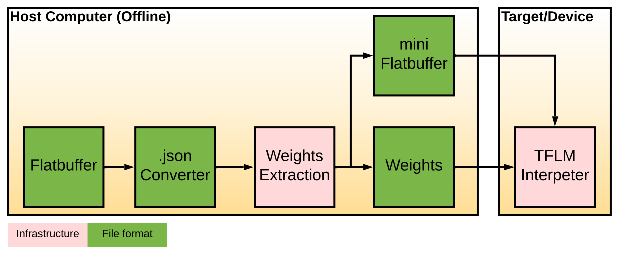

FIGURE 7. The proposed TFLite FlatBuffer conversion.

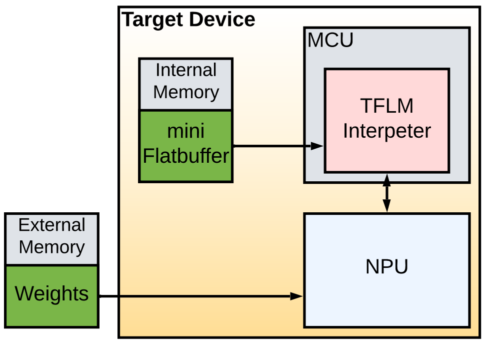

FIGURE 8. Hardware/Software TFLM proposed flow.

comprised of MCU, the proposed NPU, internal SRAM, and external memory. The network model is represented by the mini-FlatBuffer and is interpreted using the MCU, while the NPU is served as an inference accelerator for the various network layers. The compressed weights are stored in the external memory and accessed directly by the NPU when required.

IV. EXPERIMENTS AND RESULTS

The proposed methodology has been evaluated by employing the MCU-NPU acceleration for various TFLM-based Neural Network (NN) architectures. The achieved speedup is evaluated in comparison to software-only implementation, i.e., using MCU without the hardware accelerator using the original TFLM software.

In this work, the MicroBlaze processor is selected as the target MCU. The MicroBlaze processor is configured using a predefined real-time configuration [31], featuring a 16KB cache, and a tightly coupled on-chip memory, utilizing about 4000 logic cells. An external DDR memory is used for data storage.

The development environment is based on the Xilinx Embedded System Design flow and the Xilinx Vitis 2021.2. A first-order software optimization has been applied using the software GNU Compiler Collection (GCC), a common standard C compiler, with -Os flag for code size optimization. This MCU implementation is used as the baseline reference for further performance evaluations in all the following stages.

The proposed NPU module has been synthesized using a high-level synthesis (HLS) environment. The HLS enables using common C code (with additional libraries and coding styles) that is directly converted and mapped into an RTL

TABLE 1. NPU hardware accelerator utilization.

Unit	LUT	REG	BRAM	DSP	Power (mW)
AXI4-Lite	698	797	0	0	4
AXI4 Master	402	795	1	0	13
DPE	910	891	13	11	19
QAE	413	502	0	6	11
NPU (Total)	2423	2985	14	17	47
MicroBlaze	2107	2108	6	3	34
System	4530	5093	20	20	80

TABLE 2. MCU-NPU performance evaluation.

Source	Model		Runtime (ms)	
	Name	No. of FLOPS	Ref. model	MCU-NPU[6]
MLPerf [33]	KWS	5.4M	181[1]	11.6
MLPerf [33]	VWW	14.3M	603[1]	28.3
MLPerf [33]	IC	25.1M	704[1]	36.9
MLPerf [33]	AD	0.5M	10[1]	0.5
SDF [3]	GRMW	0.1M	1.7[5]	0.8
SDF [3]	HDR	5.6M	91[5]	9.9
CMSIS-NN [18]	8-bit QNN	24.6M	99.1[3]	35.3
PULP-NN [2]	8-bit QNN	24.6M	28.6[4]	20.7[7]
New	LeNet-5	1.3M	10.7[2]	2.5
New	VGG-7	29.4M	241[2]	42.4

[1] Arm-M4 (120MHz). [2] Arm-A53 (1.2GHz)
[3] Arm-M7 (216MHz). [4] Xpulp (170MHz).
[5] Nios-II/f+CIW (120MHz). [6] MCU-NPU (100MHz)
[7] MCU-NPU (170MHz)

design. The code is verified in terms of functionality versus the outputs of the TFLite software version. Table 1 shows the NPU hardware implementation utilization, the power consumption for the various NPU hardware models, and the Microblaze MCU implemented on the Xilinx Ultra96 FPGA development board with a 100 MHz clock [32]. The complete system, including the NPU and the MicroBlaze, has been synthesized using the Xilinx Vivado 2021.2 software tool. The Vivado implementation report shows that the total NPU design requires about 5, 400 logic elements, Including 2423 Look-up Tables (LUTs), 2985 Flip-Flops (REGs), 14 Block RAMs (BRAMs), and 3 Digital Signal Processors (DSPs). The NPU hardware accelerator consumes only 47 mW as derived from the Xilinx Power Estimator (XPE) tool. An appropriate NPU software kernel has been developed and added to the TFLM software library to support the proposed NPU hardware accelerator.

Table 2 depicts the performance of the proposed MCU-NPU hardware accelerator compared to nine various TFLM-based network models in terms of inference runtime, and number of Floating-Point operations (FLOPS). The nine reference network models include the four NN models of the common MLPerf Tiny benchmark [5], [33]: (a) Keyword

TABLE 3. ARM-based microNPU comparison (in 10^3 cycles).

Model	microNPU (MACs)				MCU-NPU	speedup
	1	32	64	128		
KWS	10690	344	173	93	1160	9.21
VWW	21538	805	475	366	2830	7.61
IC	25063	794	399	206	3690	6.79
AD	281	281	281	281	50	5.62
GRMW	667	26	16	14	80	8.33
HDR	6606	267	174	134	990	6.67
8-bit QNN	18955	646	323	206	3530	5.36
LeNet-5	5643	202	116	95	250	21.36
VGG-7	21435	1403	1123	1013	4240	5.05

TABLE 4. The new FlatBuffer size for the evaluated models.

Model	FlatBuffer (KB)	mini-FlatBuffer (KB)	Savings
KWS	45	23	48%
VWW	300	99	67%
IC	94	17	81%
AD	268	86	67%
GRMW	9	4	55%
HDR	102	9	91%
QNN	97	7	93%
LeNet-5	67	6	91%
VGG-7	901	12	98%

Spotting (KWS) - used to detect keywords from an audio spectrogram, (b) VisualWakeWords (VWW) - an image classification for determining the presence of a person in an image, (c) Image Classification (IC) - multi-class image classification of the CIFAR10 dataset, and (d) Anomaly Detection (AD) - used to identify anomalies in machine sounds.

To further evaluate the proposed methodology, we examined the two network models presented in [3]: (a) Google network for 'Gesture Recognition Magic Wand' (GRMW) that was trained to detect wand gestures [34], and (b) an MNIST network used for Handwritten Digit Recognition (HDR) [35]. The rest three reference network models used for comparison are (a) the model presented in both [18] and [2] for CIFAR-10 database classification and the two following popular network models, (b) LeNet-5 with the CIFAR-10 database, and (c) VGG-7 with MNIST database.

Table 2 shows that the proposed hardware accelerator MCU-NPU outperforms all reference models in terms of runtime.

Executing the model presented in [22] on the RISC-V NMU using TFLM yields a similar speedup factor (of about 50) to our proposed model for the fully-connected operations. However, while they demonstrate an overall speedup of 2.9, the proposed frameworks achieved an overall speedup of factor 4 for the same NN model. Moreover, the presented framework is limited in its ability to support generic models with different topologies, as noted in [22].

TABLE 5. Comparison of the size of the compressed pruned weights matrices.

Model	Uncompressed (KB)	Compressed (KB)	Savings
KWS	22	7.8	65%
VWW	208	70	66%
IC	77.4	25.9	66%
AD	182.3	61.4	66%
GRMW	4.3	1.8	58%
HDR	93.1	30	67%
QNN	89.4	30.1	66%
LeNet-5	61.8	20.8	66%
VGG-7	889.1	288	67%

TABLE 6. The hardware acceleration per layer (for the KWS MobileNet model).

Layer index	SW (ms)	Non-Pruned		Pruned	
		HW (ms)	speedup	HW (ms)	speedup
(1)[1]	1169	4.2	278x	1.2	974x
(2)[2]	678	2.6	260x	0.9	753x
(3,5,7)[1]	890	6	148x	1.7	519x
(4,6,8)[2]	661	2.5	264x	0.9	734x
(9)[1]	928	6	154x	1.7	541x
(10)[3]	6.1	0.3	20x	0.3	20x
(11)[4]	0.4	0.1	4x	0.03	20x
(12)[5]	5	0.1	50x	0.1	50x
Total	7478	40.3	185x	11.6	644x

[1] Conv2D. [2] DepthwiseConv2D. [3] Maxpool. [4] FullyConnected. [5] SoftMax.

The proposed NPU is also compared to the ARM-based microNPU [23]. The comparison has been performed with the ARM Cortex-M55 for the various NN models as depicted in Table 3. The ARM-based NPU can be configured with a 32/64/128 MAC units. The achieved acceleration is linear to the number of the MAC units. Since the proposed NPU has only one MAC, the microNPU results are normalized accordingly, as depicted in Table 3. A curve estimation is used for evaluating the required cycles while using a single MAC. Results show that the proposed NPU outperforms the ARM microNPU in terms of runtime by an average factor of about 8.

Each reference network model is first pre-trained using TensorFlow and then converted to TFLite using 8bit quantization of both weights and data. The weights are extracted from the FlatBuffer to provide the proposed mini-FlatBuffer and the separated weights array.

Table 4 shows the size of the FlatBuffer and the proposed mini-FlatBuffer for the nine various reference network models. Using the mini-FlatBuffer yields a memory saving of up to 98% (for the VGG-7 model) and an average saving of about 75%. The differences in the mini-FlatBuffer size results from the different network topology, derived from the number of layers and the non-extracted parameters (bias, scale and

TABLE 7. Hardware accelerator speedup for the MLPerf Tiny benchmark.

Model	MCU runtime (ms)	MCU-NPU (Non-Pruned) runtime (ms)	speedup	OPs/MHz	MCU-NPU (Pruned) runtime (ms)	speedup	OPs/MHz
KWS	7470	40.3	185x	1.34M	11.6	644x	4.65M
VWW	20510	99.4	206x	1.44M	28.3	724x	5.05M
IC	19870	132.5	149x	1.89M	36.9	538x	6.80M
AD	86.4	1.9	45x	2.78M	0.5	172x	10.00M
GRMW	232.7	3.4	68x	0.38M	0.8	290x	1.25M
HDR	5260	34.8	151x	1.62M	9.9	531x	5.65M
QNN	17530	133.1	131x	1.84M	35.3	496x	6.96M
LeNet-5	1260	9.2	137x	1.43M	2.5	504x	5.20M
VGG-7	22930	154.7	148x	1.90M	42.4	540x	6.93M

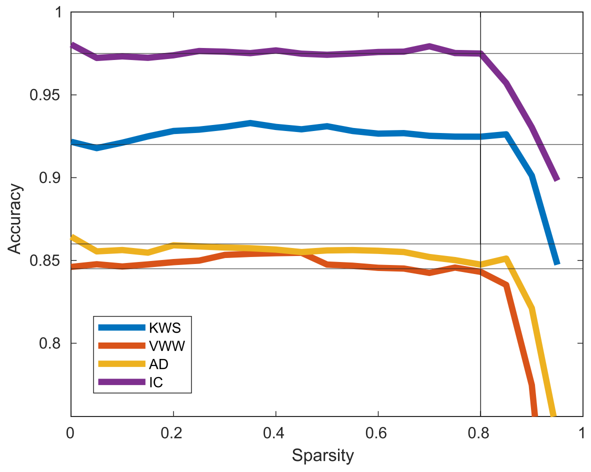

FIGURE 9. Accuracy vs. Sparsity for various MLPerf Tiny models.

offsets). Therefore, the saving factor is different for each network model.

Fig. 9 depicts the achieved accuracy as a function of various sparsity of the pruned weights matrices for the MLPerf Tiny benchmark network models. The results have been obtained using the TFLite framework and its built-in pruning tool. It can be seen that the model accuracy is kept up to the sparsity of 0.8, in accordance with the results presented in [30]. Therefore, weights pruning with 0.8 sparsity is applied for all network models for performance evaluation.

The pruned weights are further compressed using the HW-based real-time DNN lossless weight compression approach presented in [26]. Table 5 shows the size of the uncompressed and compressed pruned weights matrices. An average memory saving of about 65% is achieved for the chosen sparsity of 0.8.

Table 6 demonstrates the acceleration capabilities evaluation for the five various network layers for the KWS mobilenet network model. The results shown in Table 6 have been obtained by emulation using Xilinx FPGA. The Table shows the achieved speedup for both the pruned and non-pruned cases compared to the software-only implementation in terms of runtime. For the convolution layer (Conv2D), speedup factors of 278 and 974 are demonstrated for the non-pruned and the pruned weights, correspondingly compared to software implementation. The overall achieved speedup factor is 185 and 644 for the non-pruned and the pruned weights, correspondingly, compared to software implementation. An average speedup of about 4x is achieved while using the pruned weights with the proposed RLE encoding, demonstrating the benefit of using the proposed NPU-RLE unique module for weights zero-skipping.

Table 7 demonstrates the overall hardware accelerator speedup compared to the software model, for both pruned and non-pruned implementations, for the various evaluated network models. The table shows that the proposed hardware accelerator MCU-NPU outperforms all reference models in terms of runtime. An average speedup of 136 and 492 is achieved for the non-pruned and the pruned weights, correspondingly. A speedup factor of up to 724 is demonstrated (for VWW model), providing a total of 5.05 MOPs per MHZ.

V. CONCLUSION

This paper presents an efficient hardware/software framework to accelerate machine learning inference on edge devices using a modified TensorFlow Lite for Microcontroller (TFLM) model running on a Microcontroller (MCU) and utilizing a custom Neural Processing Unit (NPU) hardware accelerator. The new framework suggests an efficient and compact representation of the TFLM network model, separating the weights from the network model and enabling a low-memory footprint for applying the network model. In addition, the proposed framework supports weight compression of pruned quantized NN models and exploits the pruned model sparsity to reduce the computational complexity. We also suggest adding real-time decompression capabilities to the common TFLM framework by integrating an HW-based lossless DNN weight compression approach enabling on-the-fly decoding of one weight per clock cycle. The efficiency of the proposed methodology has been evaluated by employing the MCU-NPU acceleration for various TFLM-based Neural Network (NN) architectures using the common MLPerf Tiny benchmark. The proposed hardware accelerator outperforms all reference models optimized for edge devices in terms of inference runtime. Experimental results demonstrate a significant speedup of up to 724x compared to a pure software implementation.

REFERENCES

[1] T. S. Ajani, A. L. Imoize, and A. A. Atayero, "An overview of machine learning within embedded and mobile devices–optimizations and applications," *Sensors*, vol. 21, no. 13, p. 4412, Jun. 2021.

[2] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *CoRR*, vol. abs/1908.11263, Aug. 2019.

[3] E. Manor and S. Greenberg, "Using HW/SW codesign for deep neural network hardware accelerator targeting low-resources embedded processors," *IEEE Access*, vol. 10, pp. 22274–22287, 2022.

[4] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "TensorFlow lite micro: Embedded machine learning on TinyML systems," *CoRR*, vol. abs/2010.08678, pp. 1–12, Oct. 2020.

[5] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. A. Patterson, D. Pau, J. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking TinyML systems: Challenges and direction," *CoRR*, vol. abs/2003.04821, pp. 1–8, Mar. 2020.

[6] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, "Microprocessor optimizations for the Internet of Things: A survey," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 7–20, May 2018.

[7] Intel. (Mar. 2020). *Nios II Processors*. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html

[8] Xilinx. (Mar. 2020). *Microblaze Soft Processor Core*. [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html

[9] M. Stamenovic, N. L. Westhausen, L. Yang, C. Jensen, and A. Pawlicki, "Weight, block or unit? Exploring sparsity tradeoffs for speech enhancement on tiny neural accelerators," *CoRR*, vol. abs/2111.02351, pp. 1–11, Nov. 2021.

[10] ST. (2020). *Artificial Intelligence (AI) Software Expansion for STM32Cube*. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[11] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, "TinyML platforms benchmarking," *CoRR*, vol. abs/2112.01319, pp. 1–11, Nov. 2021.

[12] G. Crocioni, D. Pau, J.-M. Delorme, and G. Gruosso, "Li-ion batteries parameter estimation with tiny neural networks embedded on intelligent IoT microcontrollers," *IEEE Access*, vol. 8, pp. 122135–122146, 2020.

[13] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantizatsion and deployment of deep neural networks on microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, 2021.

[14] Microsoft. (2020). *Embedded Learning Library*. [Online]. Available: https://microsoft.github.io/ELL/

[15] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, pp. 1–12, May 2018.

[16] J. Lin, W.-M. Chen, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020, pp. 11711–11722.

[17] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "MCUNetV2: Memory-efficient patch-based inference for tiny deep learning," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2021, pp. 2346–2358.

[18] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for Arm cortex-M CPUs," 2018, *arXiv:1801.06601*.

[19] J. Vreca, K. J. X. Sturm, E. Gungl, F. Merchant, P. Bientinesi, R. Leupers, and Z. Brezocnik, "Accelerating deep learning inference in constrained embedded devices using hardware loops and a dot product unit," *IEEE Access*, vol. 8, pp. 165913–165926, 2020.

[20] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic, "The RISC-V instruction set," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2013, p. 1.

[21] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Comput. Surv.*, vol. 52, pp. 1–39, May 2019.

[22] L. Hielscher, A. Bloeck, A. Viehl, S. Reiter, M. Staiger, and O. Bringmann, "Platform generation for edge AI devices with custom hardware accelerators," in *Proc. IEEE 19th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2021, pp. 1–8.

[23] A. Skillman and T. Edso, "A technical overview of Cortex-M55 and Ethos-U55: Arm's most capable processors for endpoint AI," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2020, pp. 1–20.

[24] D. Cordesius and J. Åhlund, "Investigation of dynamic control ML algorithms on existing and future Arm microNPU systems," M.S. thesis, Dept. Elect. Inf. Technol., Fac. Eng., Lund Univ., Lund, Sweden, Jun. 2021.

[25] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Fast convolutional neural networks in low density FPGAs using zero-skipping and weight pruning," *Electronics*, vol. 8, no. 11, p. 1321, 2019.

[26] T. Malach, S. Greenberg, and M. Haiut, "Hardware-based real-time deep neural network lossless weights compression," *IEEE Access*, vol. 8, pp. 205051–205060, 2020.

[27] M. Makni, M. Baklouti, S. Niar, and M. Abid, "Performance exploration of AMBA AXI4 bus protocols for wireless sensor networks," in *Proc. IEEE/ACS 14th Int. Conf. Comput. Syst. Appl. (AICCSA)*, Oct. 2017, pp. 1163–1169.

[28] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[29] X. Yu, T. Liu, X. Wang, and D. Tao, "On compressing deep models by low rank and sparse decomposition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 67–76.

[30] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jul. 2017, pp. 1927–1934.

[31] Xilinx. (Apr. 2017). *The Microblaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp501-microblaze

[32] AVNET. (Jun. 2022). *Ultra96-V2 Board*. [Online]. Available: hhttps://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/

[33] C. Banbury *et al.*, "MLPerf tiny benchmark," in *Proc. 35th Conf. Neural Inf. Process. Syst. (NIPS)*, 2021. [Online]. Available: https://openreview.net/forum?id=8RxxwAut1BI

[34] A. Williams. (Dec. 2018). *Magic Wand Learns Spells Through Machine Learning and an IMU*. [Online]. Available: https://hackaday.com/2018/12/07/magic-wand-learns-spells-through-machine-learning-and-an-imu/

[35] D. Tassopoulos. (Jul. 2019). *Machine Learning on Embedded*. [Online]. Available: https://www.stupid-projects.com/posts/machine-learning-on-embedded-part-3/

EREZ MANOR received the B.Sc. and M.Sc. degrees in electrical and computer engineering from Ben-Gurion University, Beer-Sheva, Israel, in 2008 and 2014, respectively, where he is currently pursuing the Ph.D. degree. His research interests include AI, FPGA, and edge devices.

SHLOMO GREENBERG (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Be'er Sheva, Israel, in 1976, 1984, and 1997, respectively. He is currently an Associate Professor and the Head of the Computer Science Department, Sami Shamoon College of Engineering, and a Staff Member with the School of Electrical and Computer Engineering, Ben-Gurion University of the Negev. His main research interests include computer architecture, machine learning, image and digital signal processing, computer vision, and VLSI low-power design.