electronics

MDPI

Article

A Methodology and Open-Source Tools to Implement Convolutional Neural Networks Quantized with TensorFlow Lite on FPGAs

Dorfell Parra [1,2,*], David Escobar Sanabria [2] and Carlos Camargo [1]

[1] Department of Electrical and Electronics Engineering, Faculty of Engineering, National University of Colombia, Bogotá 111321, Colombia; cicamargoba@unal.edu.co
[2] Department of Biomedical Engineering, Lerner Research Institute, Cleveland Clinic, Cleveland, OH 44106, USA; escobad2@ccf.org
* Correspondence: dlparrap@unal.edu.co

Abstract: Convolutional neural networks (CNNs) are used for classification, as they can extract complex features from input data. The training and inference of these networks typically require platforms with CPUs and GPUs. To execute the forward propagation of neural networks in low-power devices with limited resources, TensorFlow introduced TFLite. This library enables the inference process on microcontrollers by quantizing the network parameters and utilizing integer arithmetic. A limitation of TFLite is that it does not support CNNs to perform inference on FPGAs, a critical need for embedded applications that require parallelism. Here, we present a methodology and open-source tools for implementing CNNs quantized with TFLite on FPGAs. We developed a customizable accelerator for AXI-Lite-based systems on chips (SoCs), and we tested it on a Digilent Zybo-Z7 board featuring the XC7Z020 FPGA and an ARM processor at 667 MHz. Moreover, we evaluated this approach by employing CNNs trained to identify handwritten characters using the MNIST dataset and facial expressions with the JAFFE database. We validated the accelerator results with TFLite running on a laptop with an AMD 16-thread CPU running at 4.2 GHz and 16 GB RAM. The accelerator's power consumption was 11× lower than the laptop while keeping a reasonable execution time.

Keywords: TensorFlow; TFLite; FPGA; SoC; CNN

Citation: Parra, D.; Escobar Sanabria, D.; Camargo, C. A Methodology and Open-Source Tools to Implement Convolutional Neural Networks Quantized with TensorFlow Lite on FPGAs. *Electronics* **2023**, *12*, 4367. https://doi.org/10.3390/electronics12204367

Academic Editors: Dawid Połap, Robertas Damasevicius and Hafiz Tayyab Rauf

Received: 10 September 2023
Revised: 13 October 2023
Accepted: 16 October 2023
Published: 21 October 2023

1. Introduction

Due to their ability to extract features from input data, convolutional neural networks (CNNs) are being used in machine learning (ML) applications such as object detection, facial expression recognition, and medical imaging [1–3]. The training of CNNs is typically performed on high-performance computing platforms to speed up the optimization routines determining the CNN parameters. On the other hand, the inference process (i.e., forward propagation) takes place in various hardware platforms, ranging from cloud computing to embedded systems. However, executing CNNs in embedded devices is challenging due to the power consumption and space constraints that limit their processing and memory capabilities.

Consequently, the need for more efficient neural networks has motivated the research of model compression techniques. These techniques decrease computational complexity by using fewer parameters (i.e., pruning) [4,5] or by rescaling the data representation (quantization) [6,7]. Moreover, ML frameworks have recently implemented their own pruning and quantization approaches. For instance, TensorFlow introduced TFLite (TensorFlow Lite), a library that features the quantization scheme described in [8], for performing network inference on mobile devices, microcontrollers (MCUs), and other edge devices [9].

Nonetheless, MCUs and small edge devices are not optimal for applications requiring high throughput at lower power consumption rates, characteristics inherent to field-programmable gate arrays (FPGAs). As a result, researchers have focused on speeding up the inference of CNNs on hardware using compressed networks and custom systems on chips (SoC) on FPGAs [10–15]. Overall, tools for implementing quantized CNNs on FPGA-based accelerators have the potential to advance applications that require energy efficiency, hardware flexibility, and parallelism. Additionally, utilizing FPGAs can broaden the framework's application scope by enabling the integration of ML into complex pipelines (e.g., image acquisition, pre-processing, and classification) within a single lightweight platform. This approach also permits the processing of sensitive information locally, thereby reducing the risk of data breaches and the need to employ cloud computing while keeping the cost and energy consumption attractive. Nevertheless, widely used frameworks like TensorFlow have yet to add support for FPGAs to their quantization libraries (e.g., TFLite).

In this work, we introduce an open-source methodology for implementing TFLite quantized CNNs on FPGAs. Additionally, we present an adaptable accelerator featuring IP cores designed for network inference tasks. The accelerator's architecture is compatible with the AXI-Lite interface and allows throughput or power consumption enhancements. We assessed our approach by training and quantizing two CNNs with the Modified National Institute of Standards and Technology (MNIST) dataset and the Japanese Female Facial Expression (JAFFE) dataset. We tested the accelerator by executing the network inferences on the Digilent Zybo-Z7 development board. Moreover, we validated the accelerator's outcomes by comparing them to the results obtained from TFLite running on a laptop equipped with an AMD 16-thread CPU.

2. Related Work

Recently, there have been works describing model compression techniques that decrease the computational load of neural networks. These techniques use fewer network parameters and neurons (i.e., prune) or shift their numeric representation (i.e., quantization). For instance, DeepIoT [4] compressed networks into compact, dense matrices compatible with existent libraries. Yang et al. [5] introduced an energy-aware pruning algorithm for CNNs tied to the network's consumption. Chang et al. [6] presented a mixed scheme quantization (MSQ) that combines the sum-of-power-of-2 (SP2) and fixed-point schemes. Bao et al. [7] demonstrated a learnable parameter soft clipping full integer quantization (LSFQ).

Meanwhile, many accelerators have been designed to speed up the inference of CNNs on hardware employing custom systems on chips (SoCs) on FPGAs. Zhou et al. [16] introduced a five-layer accelerator using 11-bit fixed point precision for the Modified National Institute of Standards and Technology (MNIST) digit recognition on a Virtex FPGA. Zhang et al. [17] presented a design space exploration using loop tiling to enhance the memory bandwidth. Feng et al. [18] outlined a high-throughput CNN accelerator employing fixed-point arithmetic. Xin et al. [19] proposed an optimization framework integrating an ARM processor. Guo et al. [20] leveraged bit-width partitioning of DSP resources to accelerate CNNs with FPGAs. In [14], the authors employed Wallace tree-based multipliers to replace the multiplier accumulator units (MAC) utilized in the accelerator's processing elements (PE). In [21], the authors analyzed the on-chip and off-chip memory resources and proposed a memory-optimized and energy-efficient CNN accelerator. Zhong-ling et al. [22] used various convolution parallel computing architectures to balance computing efficiency and data load bandwidth. In [2], the authors designed a high-performance task assignment framework for MPSoCs and a DPU-based accelerator. Liang et al. [1] introduced a framework that uses on-chip memory partition patterns for accelerating sparse CNNs on hardware. In [15], the authors employed the Winograd and fast Fourier transform (FFT) as fast algorithms representatives to design an architecture that reuses the feature maps efficiently.

Moreover, other works employed microcontrollers and application-specific integrated circuits (ASICs) as development platforms. Ortega-Zamorano et al. [23] described an effi-

cient implementation of the backpropagation algorithm in FPGAs and microcontrollers. In [24], the authors presented Diannao, a small-footprint, high-throughput accelerator for ML. In [25], the authors introduced Shidiannao, an integration between the Diannao accelerator and a CMOS/CCD sensor that achieved a footprint area of 4.86 mm^2. Additionally, there are surveys of neural networks on hardware that provide insights into the current state and point out the challenges that slow down the use of accelerators [10–13]. Our work supplements the existing research by introducing both a methodology for implementing TFLite-quantized CNNs in FPGAs and a customizable accelerator compatible with AXI-Lite-based SoCs.

3. Background

3.1. Model Compression

CNN parameters involved in the inference typically use 32-bit floating point numbers, which could make memory and computational demands challenging for platforms with limited resources, such as MCUs or embedded systems [10]. The aforementioned challenge has motivated the research of model compression techniques that reduce the network size. For example, pruning identifies and removes neurons that are not significantly relevant in deep networks. On the other hand, quantization re-scales the numeric range (e.g., from real numbers to integers), reducing the computational complexity of the forward propagation. Quantization could happen after training (post-training quantization) or, more effectively, during training (quantization-aware training) [8]. Moreover, in specific cases, pruning and quantization could be used together.

3.2. Quantization with TensorFlow Lite

Figure 1 describes the TFLite quantization process applied to a convolutional layer. Initially, TFLite adds the activation quantization (act quant) and weight quantization (wt quant) nodes. These nodes scale the range, making the layer aware of quantization. Following the network training, all the operations employed in the inference will use only integer numbers. For instance, Table 1 shows the specifications for the Conv_2D, Fully_Connected, and Max_Pool_2D layers, while a comprehensive list of supported operations is available in [26]. Although using integers impacts the network accuracy, it reduces the complexity of the forward propagation, particularly relevant for resource-constrained devices.

Table 1. Operator specifications of TFLite quantized layers [27].

Layer	Inputs/Outputs	Data_Type	Range
Conv_2D	Input 0:	int8	$[-128, 127]$
	Input 1 (Weight):	int8	$[-127, 127]$
	Input 2 (Bias):	int32	$[int32\_min,\ int32\_max]$
	Output 0:	int8	$[-128, 127]$
Fully_Connected	Input 0:	int8	$[-128, 127]$
	Input 1 (Weight):	int8	$[-127, 127]$
	Input 2 (Bias):	int32	$[int32\_min,\ int32\_max]$
	Output 0:	int8	$[-128, 127]$
Max_Pool_2D	Input 0:	int8	$[-128, 127]$
	Output 0:	int8	$[-128, 127]$

Furthermore, TFLite enhances the inference and training efficiency by employing custom C++ functions linked to the Python library *model_optimization* [28,29]. Essential functions

include *SaturatingRoundingDoublingHighMul*, *RoundingDivideByPOT*, and *MultiplyByQuantizedMultiplier*, whose descriptions are available in Appendix A Algorithms A1–A3 [29].

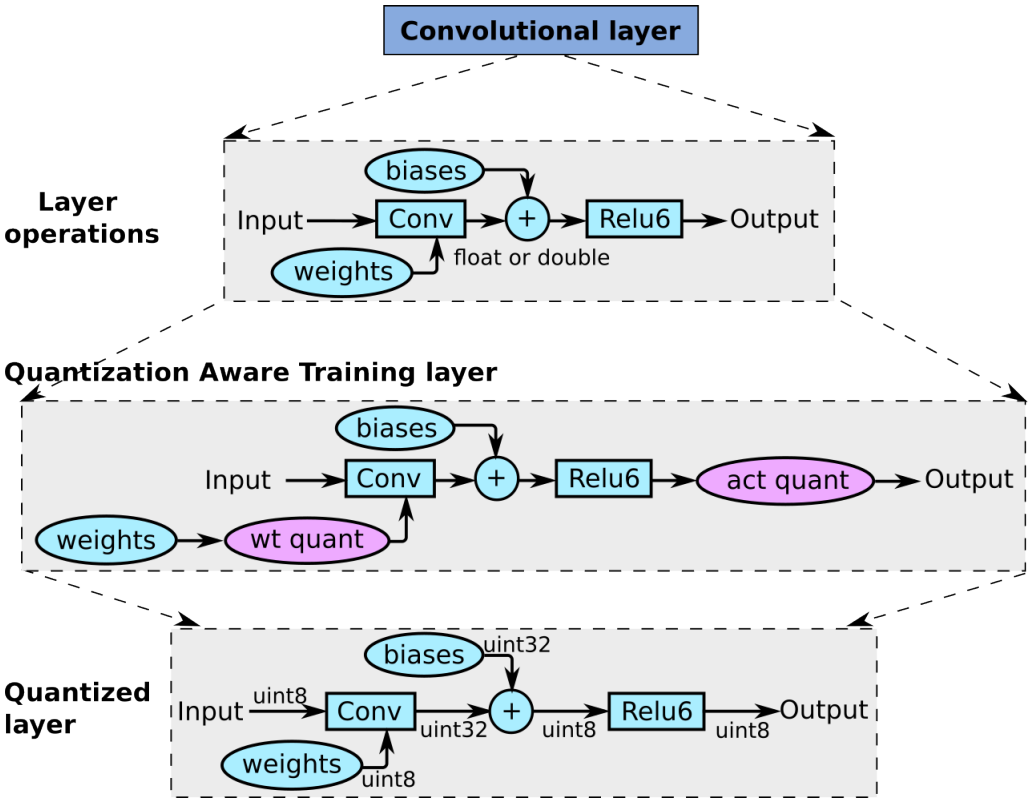


Figure 1. Example of a convolutional layer quantized with TFLite adapted from [8]. The layer operations are (1) the convolution between input and weights arrays, (2) the bias addition, and (3) the rectification via a rectified linear unit (ReLU). These operations typically employ 32-bit floating point arithmetic. The layer becomes aware of quantization after TFLite adds the weight quantization (wt quant) and activation quantization (act quant) nodes to simulate the quantization effect. After training, the inference of the quantized layer will only require integer arithmetic, making it affordable for lightweight embedded systems.

4. Materials and Methods

4.1. Accelerator Architecture

Figure 2 depicts the accelerator architecture designed for running the inference of CNN models quantized with TFLite on Zynq FPGAs [30]. The accelerator employs the Processing System 7 (PS7) to control the execution of the forward propagation, along with the custom IP cores Conv0, Mpool0, Dense0, and TFLite_mbqm0 for computing the layers' outputs.

The inference process starts with the ARM processor, controlled by the PS7, loading all the parameters and the input data from a microSD card to the system on chip's (SoC) memory. Then, the processor reads the firmware application and writes only the data needed to compute the first layer into the core wrapper registers.

Next, the core computes the operations and copies the results into the output register, which can be read by the processor and stored in RAM. These steps are repeated for all the layers in the network. This method of handling the operations enables reusing the same cores for similar layers presented in the network, as far as the temporal dependency of data allows it. All the data transfers between the ARM processor and the custom cores are made via the AXI AMBA communication bus.

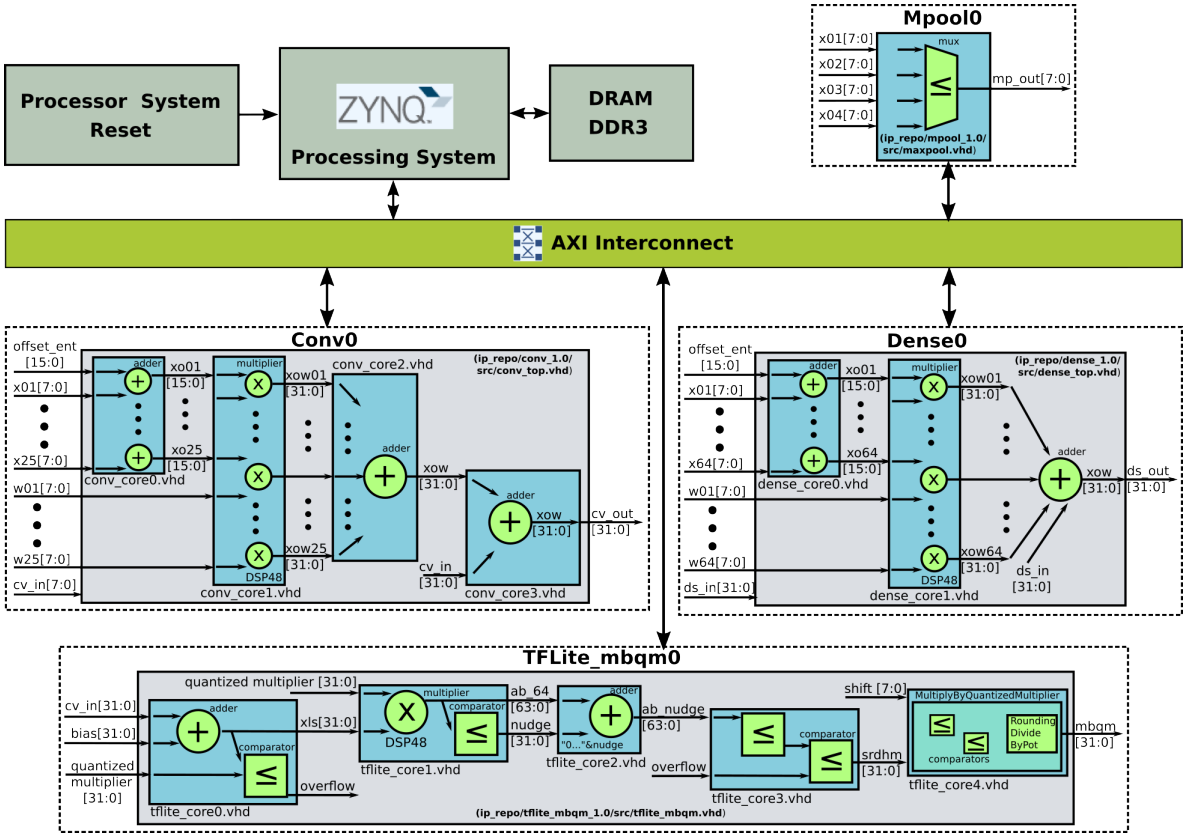


Figure 2. The accelerator architecture employs an ARM processor connected to custom IP cores via an AXI-Lite bus. Initially, the processor loads the parameters and input data into the SoC memory. Then, it transfers the data between memory and the cores' registers when needed to coordinate the inference execution. The custom IP cores support the following operations. The core TFLite_mbqm0 computes the *multiplybyquantizedmultiplier* factor used in the quantized layers. The Conv0 core calculates convolutions of 5 $\times$ 5 arrays. The Mpool0 core runs a Maxpooling operation over 2 $\times$ 2 windows. The Dense0 core takes arrays of up to 64 elements to perform the fully_connected layer. Furthermore, the cores Conv0 and Dense0 employ DSP48 resources to improve computational efficiency. Additionally, the accelerator throughput can be augmented by adding core instances and increasing the cores' size to support larger inputs.

4.1.1. TFLite_mbqm Core

The TFLite_mbqm core implements the functions *SaturatingRoundingDoublingHighMul*, *RoundingDivideByPOT*, and *MultiplyByQuantizedMultiplier* required for calculating the *mbqm* value. The core's behavior was validated with simulations, obtaining an average execution time of 600 µs. Furthermore, the function *tflite_mbqm* manages the core via an AXI-Lite wrapper, as depicted in Algorithm 1.

Algorithm 1 Computation of the value *mbqm* using the TFLite_mbqm core.

1: **function** $tflite\_mbqm(cv\_in,\ bias,\ M0,\ shift)\{$
2: **set input registers:** $cv\_in,\ bias,\ M0,\ shift$
3: $wait\ for\ the\ core\ to\ finish\ execution$
4: **get output register:** *mbqm*
5: $return\ mbqm;\}$;
6: **end function**

The core consists of five sub-modules described below:

- tflite_core0: Adds the *bias* to the input value (e.g., *cv_in*) and checks for overflow.

- tflite_core1: Multiplies the *quantized_multiplier* value with the input plus bias (*xls*), using two DSP48s because the expected result is a 64-bit width. This sub-module also computes the *nudge* variable.
- tflite_core2: Adds the *ab_64* value to the *nudge* into the *ab_nudge*.
- tflite_core3: Saturates the *ab_nudge* and bounds it to the *int32_t* maximum value. The result is the *srdhm* value.
- tflite_core4: Rounds the *srdhm* value using the *shift* parameter and outputs the *mbqm* value.

4.1.2. Conv Core

The Conv core performs TFLite-based convolutions using 5×5 kernels. Simulations were used to validate its behavior, resulting in an average execution time of 500 ns per convolution. The core comprises the following sub-modules:

- conv_core0: Adds the *offset_ent* parameter and the input values *x01, . . . , x025* into the *xo01, . . . , xo025* signals.
- conv_core1: Multiplies the weights *w01, . . . , w25* with the *xo01, . . . , xo025* values using DSP48 blocks, into *xow01, . . . , xow025* signals.
- conv_core2: Adds the *xow01, . . . , xow025* values into the signal *xow*.
- conv_core3: Adds the previous value *cv_in* to the present value *xow*. The result is stored in the output register *cv_out*.

The function *conv_k5*, depicted in Algorithm 2, is employed to compute a convolutional layer. *ent* is the input tensor with dimensions $lenX, lenY, lenZ, lenW$; *fil* is the filters tensor with dimensions $lenA, lenB, lenC, lenD$; and *cnv* is the resulting tensor with dimensions $lenA, lenB, lenC, lenD$. The parameters *shift, M0, scale, offset_ent, offset_sor* come from the quantization. The function *cv_k5_core* controls the Conv core through its base address *addr0* and the registers of its AXI-Lite wrapper. Its outputs are then directed to the TFLite_mbqm core with base address *addr1*. Following that, the first clamp operation (line 12) reproduces a ReLU from 0 to 255, while the second clamp (line 14) bounds the values to the *int8* range; *min_val* = -128 and *max_val* = 127.

Algorithm 2 Convolutional layer computation employing the Conv and TFLite_mbqm cores

```
1:  function conv_k5(ent[lenX, lenY, lenZ, lenW], fil[lenA, lenB, lenC, lenD]){
2:   define: cnv[lenE, lenF, lenG, lenH]
3:   for (f = 0; f < lenA; f++){
4:    get: shift, M0, bias
5:    for (i = 0; i < lenY − 4; i++){
6:     for (j = 0; j < lenZ − 4; j++){
7:      for (k = 0; k < lenW; k++){
8:       cnv[0][i][j][f] = cv_k5_core(addr0,
9:                                    ent[0, 0 + i, 0 + j, k], . . . , fil[f, 0, 0, k], . . . ,
10:                                   offset_ent, cnv[0][i][j][f])};
11:     cnv[0][i][j][f] = tflite_mbqm(addr1, cnv[0][i][j][k], bias, M0, shift);
12:     cnv[0][i][j][f] = min (max(cnv[0][i][j][f], 0), 255);
13:     cnv[0][i][j][f] = cnv[0][i][j][f] + offset_sor;
14:     cnv[0][i][j][f] = min (max(cnv[0][i][j][f], −128), 127);
15:  }; }; };
16:  return cnv;};
17: end function
```

4.1.3. Mpool Core

The Mpool core takes four values and returns the maximum, utilizing an AXI-Lite wrapper controlled by the function *mp_22_core*. Algorithm 3 presents the function *maxp_22* used to compute a MaxPooling layer using 2×2 windows over the input data. The input tensor *cnv* has dimensions *lenX*, *lenY*, *lenZ*, and *lenW*, and the resulting tensor named *mxp* has dimensions *lenA*, *lenB*, *lenC*, and *lenD*.

Algorithm 3 Maxpooling layer computation using the Mpool core.

```
1: function maxp_22(cnv[lenX, lenY, lenZ, lenW]){
2:   define mxp[lenA, lenB, lenC, lenD]
3:   for (i = 0; i < lenB; i++){
4:     for (j = 0; j < lenC; j++){
5:       for (k = 0; k < lenD; k++){
6:         mxp[0][i][j][k] = mp_22_core(addr,
7:           cnv[0, 0 + i * 2, 0 + j * 2, k], cnv[0, 0 + i * 2, 1 + j * 2, k],
8:           cnv[0, 1 + i * 2, 0 + j * 2, k], cnv[0, 1 + i * 2, 1 + j * 2, k]);
9:   }; }; };
10:  return mxp};
11: end function
```

4.1.4. Dense Core

The Dense core performs the computationally intensive operations of a TFLite-based fully connected layer with vectors of up to sixty-four elements. The core's behavior was validated using simulation, yielding an estimated execution time of 500 ns. The core consists of the following sub-modules:

- dense_core0: Adds the *offset_ent* parameter and the input values *x01, . . . , x025*, and copies the results into the *xo01, . . . , xo025* signals.
- dense_core1: Multiplies the weights *w01, . . . , w025* by the *xo01, . . . , xo025* values using DSP48, and copies the results into the *xow01, . . . , xow025* signals. Then, these signals are added in the top module, and the result is stored in the output register *ds_out*.

The function *dense*, described in Algorithm 4, is employed to compute a fully connected layer. *ent* is the input vector of size $lenX$; *fil* is the filters matrix with dimensions $lenY$, $lenZ$; and *dns* is the resulting vector of size $lenW$. The parameters $shift$, $M0$, $scale$, $offset\_ent$, $offset\_sor$ are derived from the quantization process. The Dense core is controlled by employing its base address *addr0* and its AXI-Lite wrapper, managed by the function *ds_k64_core*. Its outputs are then directed to the TFLite_mbqm core with base address *addr1*. Next, the *output_offset* is added, and the results are bounded by the *int8* range (line 11).

Algorithm 4 Fully connected layer computation using the Dense and the TFLite_mbqm cores

```
1: function dense(ent[lenX], fil[lenY, lenZ]){
2:   define dns[lenW]
3:   for (f = 0; f < lenY; f++){
4:     get: shift, M0, bias
5:     for (i = 0; i < lenX/64; i++){
6:       dns[f] = ds_k64_core(addr0, offset_ent, dns[f],
7:                            ent[0 + 64 * i], ..., ent[63 + 64 * i],
8:                            fil[f][0 + 64 * i], ..., fil[f][63 + 64 * i]); };
9:     dns[f] = tflite_mbqm(addr1, dns[f], bias, M0, shift);
10:    dns[f] = dns[f] + offset_sor;
11:    dns[f] = min (max(dns[f], −128), 127); };
12:  return dns; };
13: end function
```

4.1.5. Additional Functions

Additional functions that support the inference process are *padding* and *flatten*. The *padding* function, described in Algorithm 5, is in charge of introducing zero-value elements to the tensor. This maintains the size consistency between the input and output tensors of the layer. *ent* is the input tensor, and *pad* is the output tensor with dimensions $lenX$, $lenY$, $lenZ$, $lenW$ and $lenA$, $lenB$, $lenC$, $lenD$, respectively. Furthermore, because quantization shifts the zero position, the new value is given by the *zero_point* parameter.

Algorithm 5 Padding computation for quantized network.

```
function padding(ent[lenX, lenY, lenZ, lenW], zero_point,
                 pad[lenA, lenB, lenC, lenD]){
 for (f = 0; f < lenX; f++){
  for (i = 0; i < lenY; i++){
   for (j = 0; j < lenZ; j++){
    for (k = 0; k < lenW; k++){
     if (outside input tensor boundaries){
      pad[f,i,j,k] = zero_point;};
     else{
      pad[f,i,j,k] = ent[f,i-2,j-2,k];};
 }; }; }; }; };
end function
```

The *flatten* function, described in Algorithm 6, takes a tensor and creates its 1D array. *ent* is the input tensor with dimensions $lenX$, $lenY$, $lenZ$. *flt* is the output vector whose dimensions depend on the number of characteristic maps, their sizes, and the number of classes.

Algorithm 6 Flatten function.

```
function flatten(ent[lenX, lenY, lenZ]){
 define flt[lenX × lenY × lenZ], int idx = 0;
 for (i = 0; i < lenX; i++){
  for (j = 0; j < lenY; j++){
   for (k = 0; k < lenZ; k++){
    flt[idx] = ent[0,i,j,k];
    idx += 1;
 }; }; }; };
end function
```

4.2. Methodology Overview

The proposed methodology is described in Algorithm 7. From the hardware perspective, the user needs to provide a pre-processed dataset, a Zynq FPGA platform, and a CNN quantized with TFLite. Then, the trained parameters and the input data need to be copied to a microSD card. The next step involves exporting the accelerator's hardware specification file (*.xsa) from Vivado to Vitis. Nevertheless, if needed, this hardware–software architecture can be customized by adding more core instances or modifying the kernel sizes to enhance resource utilization, throughput, and power.

From the software perspective, the user maps the quantized network onto the accelerator through a C application. Algorithm 8 depicts the Vitis template we developed, and it is described as follows. First, the function *network_inference* retrieves the input data (InTensor), the parameters (ParMatrix), and the filters (FilTensor). Next, padding is applied to keep the layers' size, and then the user adds the network layers to the template. After compilation, the FPGA can execute the inference of the quantized network.

Figure 3 provides an overview of the proposed methodology, including the quantized network, the functions for executing the layers' computation and controlling the IP cores, and the accelerator's memory map and architecture.

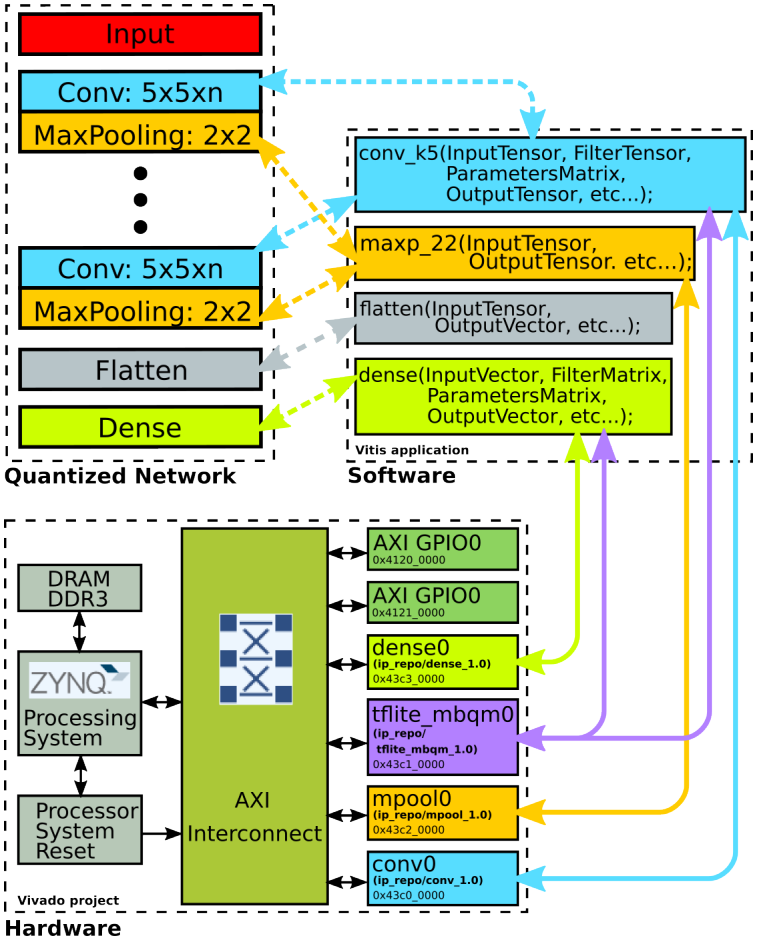


Figure 3. The proposed methodology allows for running the inference of CNNs quantized with TFLite on FPGAs. The color arrows depict the relationship between the quantized layers, IP cores, and handle functions. Specifically, the dotted arrows show the connection between the layers and the functions, while the solid ones show what functions manage the hardware cores. After training and quantizing the model, the user maps the CNN into the accelerator using the Vitis application we provided. Additionally, the Vivado project supplies the hardware description files required to customize the accelerator, while Vitis imports its hardware specification from Vivado to link it to the C application. After programming the FPGA, the inference can be executed and monitored via a serial terminal. All the files involved in the methodology are available in the open-source repository https://gitlab.com/dorfell/fer_sys_dev (accessed on 9 September 2023).

Algorithm 7 Methodology to implement quantized CNNs in Zynq FPGAs

Require:
  Pre-processed data set.
  Zynq FPGA platform.
Ensure:
  CNN trained and quantized.
  Network parameters in microSD card.
  Hardware files (∗.xsa, ∗.bitstream) from Vivado.
  Map the network in Vitis.
Execute:
  Compile project and program FPGA.
  Open Serial terminal to control the execution.
  Run inference.
Assessment:
  Accuracy, loss, execution time, etc.

Algorithm 8 C application template for mapping TFLite quantized CNNs in Vitis

```
function network_inference(InTensor, ParMatrix, FilTensors){
 define: PadTensors
 add layers:
  padding(InTensor, PadTensor, zero_point);
  conv_k5(PadTensor, FilTensor, ParMatrix, CnvTensor);
  maxp_22(CnvTensor, MxpTensor);
  * * *
  flatten(MxpTensor, FltVector);
  * * *
  dense(FltVector, FilMatrix, ParMatrix, DnsVector);
 get output: DnsVector
 return 0; };
end function
```

4.3. Experimental Setup

We assessed our methodology by employing two CNNs quantized through TFLite and trained on two datasets: the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits [31] and the Japanese Female Facial Expression (JAFFE) dataset [32]. We chose the MNIST dataset because it is a classification benchmark for ML algorithms with 70,000 images of handwritten numbers from zero to nine. Conversely, we selected JAFFE because it is employed in the more challenging facial expression recognition (FER) task. This dataset comprises 213 images of ten female subjects performing six basic facial expressions plus a neutral one. The accelerator's synthesis was carried out using Vivado (v2021.1), and the application compilation for the network inference utilized Vitis (v2021.1). Our tests utilized the Zybo-Z7 development board made by Digilent, featuring a Xilinx FPGA Zynq XC7Z020 with an ARM CPU processor operating at 660 MHz and 1 GB of RAM. To understand how our FPGA hardware's execution time and power consumption compared to traditional computer architectures, we used a Legion 5 laptop equipped with an AMD Ryzen7 4800H 16-core CPU at 4.2 GHz and 16 GB of RAM.

5. Results

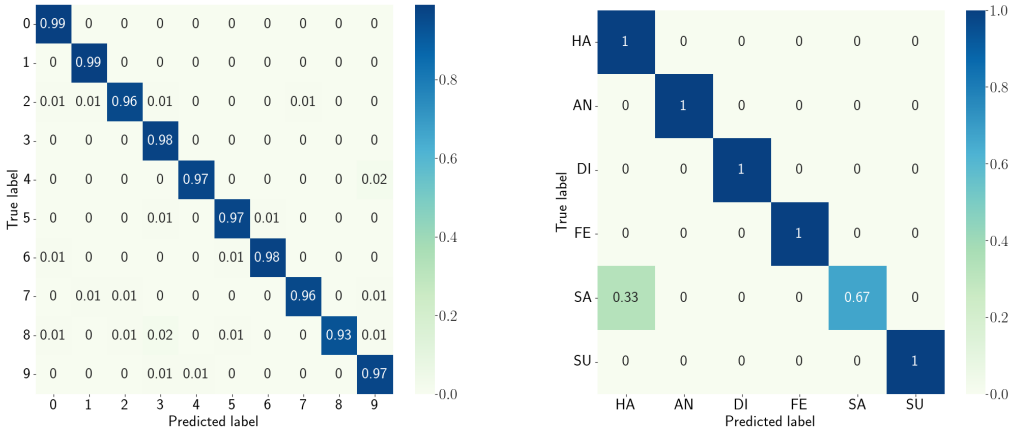
5.1. Trained Models

We trained our first CNN with MNIST (CNN+MNIST) using the example provided in [28]. This model employed a convolutional layer of kernel size 5×5 with five filters, a pooling layer, and a dense layer with ten neurons. For the CNN trained with the JAFFE dataset (CNN+JAFFE), due to the complexity of FER tasks, we implemented a pre-processing pipeline (i.e., detection of eyes, rotation of face, cropping the region of interest, and equalizing the image histogram) following the methodology outlined in [33]. Additionally, we enhanced the pre-processed dataset using the local binary pattern (LBP) descriptor. Then, we improved the robustness of the training by employing data augmentation to generate up to fifteen new samples from each original image. This model used three convolutional layers of kernel size 5×5 with 32, 64, and 128 filters, pooling layers, and a dense layer with six neurons. Table 2 summarizes these models' architectures. The confusion matrix shown in Figure 4a shows that the CNN+MNIST successfully classified the dataset. However, the confusion matrix for the CNN+JAFFE, presented in Figure 4b, indicates that the network is overfitting. This behavior can happen when the number of trainable parameters is not optimal and the network fails to generalize the dataset. Furthermore, Table 2 shows the precision, recall, F1-score, Matthews correlation coefficient (MCC), and accuracy metrics achieved by the two CNNs. At first glance, the performance of the CNN+JAFFE is outstanding, but we know from the confusion matrix that this is not the case. Therefore, every metric value should be analyzed per class to better understand how the network performs. Of note, the primary purpose of using these networks as examples

is to validate our proposed methodology for implementing CNNs quantized with TFLite on lightweight FPGAs, not to optimize their performance.

Table 2. The classification metrics precision, recall, F1-score, Matthews correlation coefficient (MCC), and accuracy obtained by the models trained with the MNIST and JAFFE datasets.

Name	Model	Precision	Recall	F1-Score	MCC	Accuracy
CNN+ MNIST	Input: 28 × 28 Conv2D: 5 × 5 × 5 MaxPooling: 2 × 2 Dense: 10	97.15%	97.11%	97.11	97.12%	96.81%
CNN+ JAFFE	Input: 64 × 64 Conv2D: 32 × 5 × 5 MaxPooling: 2 × 2 Conv2D: 64 × 5 × 5 MaxPooling: 2 × 2 Conv2D: 128 × 5 × 5 MaxPooling: 2 × 2 Dense: 6	95.83%	94.44%	93.78%	94.28%	93.78%



(**a**) (**b**)

Figure 4. Confusion matrices of the CNNs employed to assess the methodology. (**a**) CNN+MNIST: Its classes correspond to handwritten digits from zero to nine. Overall, the network trained with MNIST successfully classified all the test samples. (**b**) CNN+JAFFE: Its classes are six emotions (i.e., happiness (HA), anger (AN), disgust (DI), fear (FE), sadness (SA), and surprise (SU)), represented with facial expressions. The resulting overfitting indicates that the network trained with JAFFE struggles to generalize the dataset.

5.2. Quantized Models

Table 3 presents the accuracy, number of parameters, and size of the two convolutional networks before and after quantization. The CNN+MNIST achieved a 96.79% accuracy, using 9.940 parameters and a size of 40.64 kB. Once the model was quantized, the number of parameters increased to 9.964, while the accuracy and size dropped to 94.43% and 13.30 kB, respectively. On the other hand, the CNN+JAFFE model obtained an accuracy of 94.44% employing 306.182 parameters and with a size of 1.17 MB. After quantization, the number of parameters rose to 306.652, and the accuracy and size decreased to 83.33% and 0.30 MB. The performance drop observed after quantization can be attributed to loss of information

caused by shrinking the parameters representation from the floating-point range to a fixed number set [28]. For instance, the numbers 1.0, 1.1, and 1.2 might all be represented by the same value during quantization (e.g., 1.0), creating a lossy parameter. Using these lossy parameters in intensive calculations can lead to accumulating numerical errors and propagating them in subsequent computations.

Table 3. Models' numerical representation, accuracy, number of parameters, and size before and after quantization.

Name	Model	Representation	Accuracy	Parameters	Size
CNN+ MNIST	Input: 28 × 28 Conv2D: 5 × 5 × 5 MaxPooling: 2 × 2 Dense: 10	Floating Point	96.79%	9.940	40.64 kB
		Integer	94.43%	9.964	13.30 kB
CNN+ JAFFE	Input: 64 × 64 Conv2D: 32 × 5 × 5 MaxPooling: 2 × 2 Conv2D: 64 × 5 × 5 MaxPooling: 2 × 2 Conv2D: 128 × 5 × 5 MaxPooling: 2 × 2 Dense: 6	Floating Point	94.44%	306.182	1.17 MB
		Integer	83.33%	306.652	0.30 MB

5.3. Logic Resources and Power Consumption

Figure 5 shows the accelerator's placement and routing within the FPGA, and Table 4 presents the logic resources employed. Although adding more core instances to the data path improves throughput, the area of the device did not allow it. Furthermore, Figure A1 displays the Vivado estimation of the power consumption. The ARM processor uses about 1.53 W of power, while the total estimated power is less than 1.7 W. Notably, this is nearly 3× lower than the laptop's power consumption in the idle state [34].

Table 4. Utilization of logic resources.

Resource	Available	Utilization	Utilization %
LUT	53,200	6373	11.98
LUTRAM	17,400	71	0.41
FF	106,400	12,470	11.72
DSP	220	93	42.27
IO	125	18	14.40

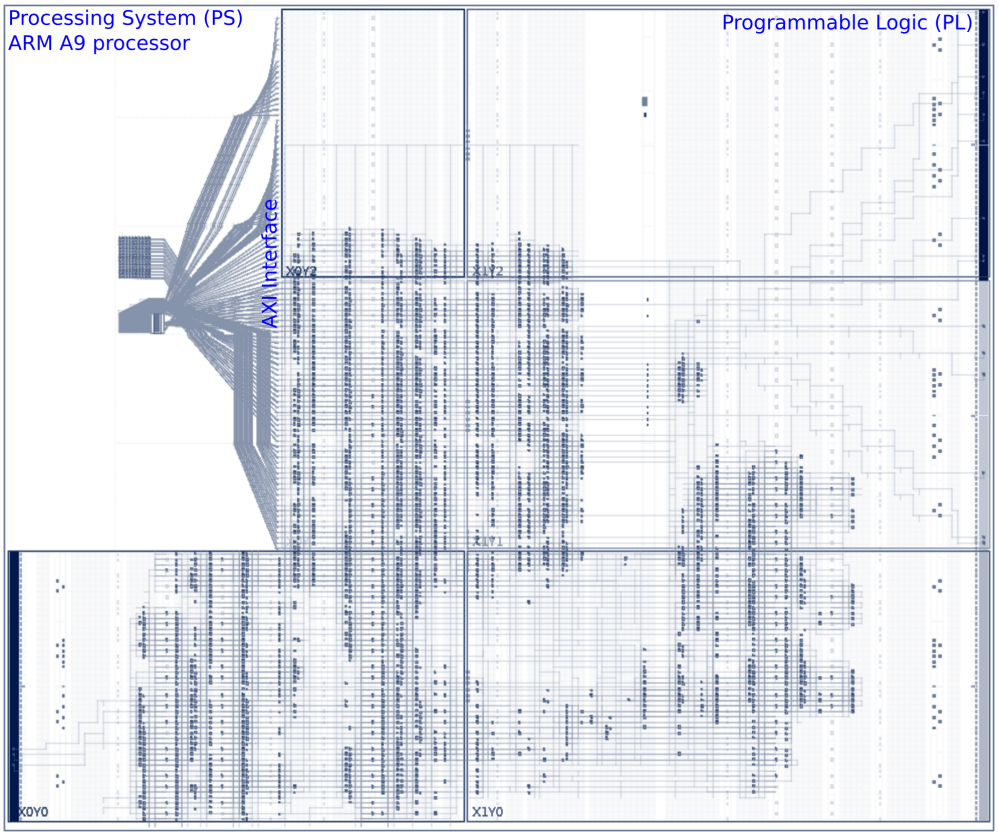


Figure 5. A Zynq FPGA combines a processing system (PS) with programmable logic (PL). Typically, the PS is a hardcore ARM processor with one or more cores. Meanwhile, the PL encompasses the devices' logic resources, BRAMs, DSP48, and I/O buffers. These resources are organized in slices, identified with XY coordinates. The place and route stage of the Vivado design flow implements the accelerator and the data path on the FPGA employing the PL. For our performance evaluation, we utilized a Zybo-Z7 board equipped with the XC7Z020 device. While some slices were partially utilized, the resources required for the data path made adding more core instances unfeasible.

5.4. Performance Comparison

The accelerator's performance was compared against a laptop running the inference of the two quantized networks CNNs+MNIST and CNN+JAFFE. The accelerator executed the inference of the networks employing a bare-metal C application. Meanwhile, a laptop with Ubuntu 20.04.2 LTS utilized the TFLite included in TensorFlow version 2.6.0.

Table 5 presents the models' accuracy and inference times on the tested platforms. Here, it is relevant to point out that for the Zybo-Z7, the reported inference times do not consider the firmware compilation in Vitis. The CNN+MNIST achieved an accuracy of 94.43% employing 9.964 parameters after quantization, and its inference on the accelerator was 35× faster than the laptop. Conversely, the CNN+JAFFE obtained a post-quantization accuracy of 83.33% utilizing 306.652 parameters, but the accelerator performance was 1.35× slower. This slowdown indicates a computation bottleneck caused by using a single Conv core for processing three layers with 32, 64, and 128 filters. This deceleration was not observed with the CNN+MNIST because that model only had one convolutional layer with five filters. Moreover, memory bottlenecks can be ruled out because a maximum of 64 elements were transferred simultaneously from the SoC's memory to the cores' registers. While the resources available on the Zybo-Z7 FPGA limited the number of cores in our implementation, the accelerator can handle more core instances to enhance performance if a larger FPGA is used.

Additionally, it is worth noting that the accelerator power consumption required only 4.5 W, whereas the laptop required around 50 W. These factors and cost considerations make our implementation a compelling choice for battery-powered remote applications.

Table 5. Comparison of model inferences on laptop and Zybo-Z7.

Quantized Network	Platform	Accuracy	Inference Time	Power	Cost
CNN+MNIST	Laptop with TFLite	94.43%	4.45 s	50 W	$950
CNN+JAFFE		83.33%	73.97 s		
CNN+MNIST	Zybo-Z7 with C application	94.43%	0.127 s	4.5 W	$299
CNN+JAFFE		83.33%	99.74 s		

6. Discussion and Conclusions

In this work, we introduced and validated an open-source methodology for running the inference of quantized CNNs on Zynq FPGAs. Initially, we employed TensorFlow to train two CNNs with the MNIST (CNN+MNIST) and the JAFFE (CNN+JAFFE) datasets. We used confusion matrices and the precision, recall, F1-score, Matthews correlation coefficient (MCC), and accuracy metrics to assess their classification performance. While the CNN+MNIST successfully classified the dataset, the confusion matrix for the CNN+JAFFE showed that the network struggled to generalize the dataset. We addressed this overfitting by using data augmentation. However, to ensure the model remained suitable for lightweight FPGAs, we refrained from enlarging it by adding dropout layers or mixing it with other ML algorithms concurrently.

Then, we employed TFLite to quantize the networks, resulting in a decrease in accuracy of 2.36% and 11.11% for the CNN+MNIST and CNN+JAFFE networks, respectively. This drop in performance reflects information loss and propagation of numerical error through the network caused by shifting the float-point representation for integer numbers. Nonetheless, utilizing integer arithmetic is still attractive because it reduces the computational load associated and renders the network inference feasible for devices with limited resources.

Additionally, we developed an adaptable accelerator compatible with the AXI-Lite bus and enhanced it with DPS48 and BRAMs resources through synthesis primitives. We also provided the hardware description language (HDL) design files to customize the architecture (e.g., by varying the size of concurrent operations or modifying the number of IP core instances). Moreover, we made the C application template needed for mapping the CNNs into the accelerator available and evaluated our methodology with a Digilent Zybo-Z7 FPGA platform.

The experiments showed that compared with a Legion 5 laptop, our accelerator achieved a 35× increase in speed for the MNIST CNN but was 1.35× slower with the JAFFE CNN. We attribute this slowdown to a computational bottleneck caused by using a single Conv core for processing three layers with 32, 64, and 128 filters. For the CNN+MNIST, the computational bottleneck was negligible because it only had one convolutional layer with five filters. Furthermore, memory bottlenecks were not an issue since the maximum number of elements transferred simultaneously between the SoC memory and the cores' registers is 64. Conversely, the energy efficiency improved by 11×, making the accelerator suitable for cost-effective, battery-powered applications that require parallel computing.

Overall, this work extends the use of CNNs to applications where computational loads make edge devices unfeasible because it provides an open-source accelerator compatible

with any SoC with AXI interface support. The accelerator executes models with Conv, Max-pooling, and Dense TFLite layers on FPGAs, allowing the user to customize its accelerator architecture. Nevertheless, for complex ML models that require faster FPGAs with large memory and high throughput while being energy-efficient, advanced devices like MPSoCs with deep processing units (DPU) on Zynq Ultrascale FPGAs are advisable.

Author Contributions: D.P.: Conceptualization, Methodology, Hardware, Software, Validation, Writing—original draft. D.E.S.: Supervision, Writing—review. C.C.: Conceptualization, Supervision, Writing—review. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by a fellowship provided to D.P. by the Universidad Nacional de Colombia and also supported by the seed funds provided to D.E.S. by the Department of Biomedical Engineering at the Cleveland Clinic Lerner Research Institute.

Data Availability Statement: The data presented in this study are available in https://gitlab.com/dorfell/fer_sys_dev (accessed on 9 September 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CNN	Convolutional neural network
CPU	Central processing unit
DPU	Deep processing unit
DSP	Digital signal processor
FPGA	Field-programmable gate array
GPU	Graphics processing unit
HDL	Hardware description language
JAFFE	Japanese Female Facial Expression
LBP	Local binary pattern
MCU	Microcontroller unit
ML	Machine learning
MNIST	Modified National Institute of Standards and Technology database
TFLite	TensorFlow Lite

Appendix A. TFLite Functions Used in Quantization

Algorithm A1 *SaturatingRoundingDoublingHighMul* saturates the product between the input value (a) and the quantized_multiplier (b) and bounds its output to the int32_t maximum.

```
1:  function SaturatingRoundingDoublingHighMul (int32_t a, int32_t b) {
2:    bool overflow = a == b && a == numeric_limits < int32_t > min();
3:    int64_t a_64(a); int64_t b_64(b);
4:    int64_t ab_64 = a_64 * b_64;
5:    int32_t nudge = ab_64 >= 0 ? (1 << 30) : (1 - (1 << 30));
6:    int32_t ab_x2_high32 =
7:      static_cast < int32_t > ((ab_64 + nudge) / (1ll << 31));
8:    return overflow ? numeric_limits < int32_t > max() :
9:      ab_x2_high32; };
10: end function
```

Algorithm A2 *RoundingDivideByPOT* rounds the saturated value employing the exponent parameter and the functions BitAnd, MaskIfLessThan, MaskIfGreaterThan, and ShiftRight.

```
1:  function RoundingDivideByPOT (int32_t x, int8_t exponent) {
2:    assert(exponent >= 0);
3:    assert(exponent <= 31);
4:    const int32_t mask = Dup((1ll << exponent) − 1);
5:    const int32_t zero = Dup(0);
6:    const int32_t one = Dup(1);
7:    const int32_t remainder = BitAnd(x, mask);
8:    const int32_t threshold =
9:      Add (ShiftRight(mask, 1), BitAnd(MaskIfLessThan(x, zero), one));
10:   return Add (ShiftRight(x, exponent),
11:               BitAnd (MaskIfGreaterThan(remainder, threshold), one)); };
12: end function
```

Algorithm A3 *MultiplyByQuantizedMultiplier* calls the above functions and uses the exponent obtained with the shift quantization parameter to compute the *mbqm* factor.

```
1:  function MultiplyByQuantizedMultiplier (int32_t x
2:    int32_t quantized_multiplier, int shift) {
3:    int8_t left_shift = shift > 0? shift : 0;
4:    int8_t right_shift = shift > 0? 0 : −shift;
5:    return RoundingDivideByPOT(
6:      SaturatingRoundingDoublingHighMul (
7:        x ∗ (1 << left_shift), quantized_multiplier), right_shift); };
8:  end function
```

Appendix B. Accelerator Power Consumption

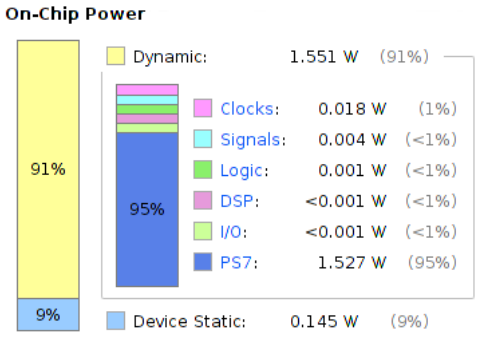


Figure A1. The power consumption estimation of the accelerator implemented on the Zynq XC7Z020 FPGA is around 2 W, making our design attractive for battery-powered applications.

References

1. Liang, Y.; Lu, L.; Xie, J. OMNI: A Framework for Integrating Hardware and Software Optimizations for Sparse CNNs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *40*, 1648–1661. [CrossRef]
2. Zhu, J.; Wang, L.; Liu, H.; Tian, S.; Deng, Q.; Li, J. An Efficient Task Assignment Framework to Accelerate DPU-Based Convolutional Neural Network Inference on FPGAs. *IEEE Access* **2020**, *8*, 83224–83237. [CrossRef]
3. Sarvamangala, D.R.; Kulkarni, R.V. Convolutional neural networks in medical image understanding: A survey. *Evol. Intell.* **2022**, *15*, 1–22. [CrossRef] [PubMed]
4. Yao, S.; Zhao, Y.; Zhang, A.; Su, L.; Abdelzaher, T. DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework. 2017. Available online: https://arxiv.org/abs/1706.01215 (accessed on 9 September 2023).
5. Yang, T.J.; Chen, Y.H.; Sze, V. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. 2017. Available online: http://xxx.lanl.gov/abs/1611.05128 (accessed on 9 September 2023).
6. Chang, S.E.; Li, Y.; Sun, M.; Shi, R.; So, H.K.H.; Qian, X.; Wang, Y.; Lin, X. Mix and Match: A Novel FPGA-Centric Deep Neural Network Quantization Framework. 2020. Available online: http://xxx.lanl.gov/abs/2012.04240 (accessed on 9 September 2023).
7. Bao, Z.; Fu, G.; Zhang, W.; Zhan, K.; Guo, J. LSFQ: A Low-Bit Full Integer Quantization for High-Performance FPGA-Based CNN Acceleration. *IEEE Micro* **2022**, *42*, 8–15. [CrossRef]

8. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficitent Integer-Arithmetic-Only Inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–22 June 2018; pp. 2704–2713.
9. TensorFlow. TensorFlow for Mobile and Edge. Available online: https://www.tensorflow.org/lite (accessed on 9 September 2023).
10. Merenda, M.; Porcaro, C.; Iero, D. Edge Machine Learning for AI-Enabled IoT Devices: A Review. *Sensors* **2020**, *20*, 2533. [CrossRef] [PubMed]
11. Misra, J.; Saha, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing* **2010**, *74*, 239–255. [CrossRef]
12. Maloney, S. Survey: Implementing Dense Neural Networks in Hardware. 2013. Available online: https://pdfs.semanticscholar.org/b709/459d8b52783f58f1c118619ec42f3b10e952.pdf (accessed on 15 February 2018).
13. Krizhevsky, A. Survey: Implementing Dense Neural Networks in Hardware. 2014. Available online: https://arxiv.org/abs/1404.5997 (accessed on 15 February 2018).
14. Farrukh, F.U.D.; Xie, T.; Zhang, C.; Wang, Z. Optimization for Efficient Hardware Implementation of CNN on FPGA. In Proceedings of the 2018 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA), Beijing, China, 21–23 November 2018; pp. 88–89. [CrossRef]
15. Liang, Y.; Lu, L.; Xiao, Q.; Yan, S. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 857–870. [CrossRef]
16. Zhou, Y.; Jiang, J. An FPGA-based accelerator implementation for deep convolutional neural networks. In Proceedings of the 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, China, 19–20 December 2015; Volume 1, pp. 829–832. [CrossRef]
17. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 22–24 February 2015; FPGA '15; pp. 161–170. [CrossRef]
18. Feng, G.; Hu, Z.; Chen, S.; Wu, F. Energy-efficient and high-throughput FPGA-based accelerator for Convolutional Neural Networks. In Proceedings of the 2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), Hangzhou, China, 25–28 October 2016; pp. 624–626. [CrossRef]
19. Li, X.; Cai, Y.; Han, J.; Zeng, X. A high utilization FPGA-based accelerator for variable-scale convolutional neural network. In Proceedings of the 2017 IEEE 12th International Conference on ASIC (ASICON), Guiyang, China, 25–28 October 2017; pp. 944–947. [CrossRef]
20. Guo, J.; Yin, S.; Ouyang, P.; Liu, L.; Wei, S. Bit-Width Based Resource Partitioning for CNN Acceleration on FPGA. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; p. 31. [CrossRef]
21. Chang, X.; Pan, H.; Zhang, D.; Sun, Q.; Lin, W. A Memory-Optimized and Energy-Efficient CNN Acceleration Architecture Based on FPGA. In Proceedings of the 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), Vancouver, BC, Canada, 12–14 June 2019; pp. 2137–2141. [CrossRef]
22. Zong-ling, L.; Lu-yuan, W.; Ji-yang, Y.; Bo-wen, C.; Liang, H. The Design of Lightweight and Multi Parallel CNN Accelerator Based on FPGA. In Proceedings of the 2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC), Chongqing, China, 24–26 May 2019; pp. 1521–1528. [CrossRef]
23. Ortega-Zamorano, F.; Jerez, J.M.; Munoz, D.U.; Luque-Baena, R.M.; Franco, L. Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers. *IEEE Trans. Neural Netw. Learn. Syst.* **2016**, *27*, 1840–1850. [CrossRef] [PubMed]
24. Chen, T.; Du, Z.; Sun, N.; Wang, J.; Wu, C.; Chen, Y.; Temam, O. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, San Diego, CA, USA, 27 April–1 May 2014; ASPLOS '14; pp. 269–284. [CrossRef]
25. Du, Z.; Fasthuber, R.; Chen, T.; Ienne, P.; Li, L.; Luo, T.; Feng, X.; Chen, Y.; Temam, O. ShiDianNao: Shifting vision processing closer to the sensor. In Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, Oregon, 13–17 June 2015; pp. 92–104. [CrossRef]
26. TensorFlow: An Open-Source Software Library for Machine Intelligence. Available online: https://www.tensorflow.org/ (accessed on 15 February 2018).
27. TensorFlow. TensorFlow Lite 8-Bit Quantization Specification. Available online: https://www.tensorflow.org/lite/performance/quantization_spec (accessed on 28 January 2022).
28. TensorFlow. Quantization Aware Training. Available online: https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html (accessed on 28 January 2022).
29. TensorFlow. TensorFlow TFLite-Micro. 2023. Available online: https://github.com/tensorflow/tflite-micro/tree/main (accessed on 11 July 2023).
30. Xilinx. Zynq Ultrascale+ MPSoC. Available online: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (accessed on 12 September 2022).
31. LeCun, Y.; Cortes, C.; Burges, C. MNIST Handwritten Digit Database. ATT Labs [Online]. 2010. Volume 2. Available online: http://yann.lecun.com/exdb/mnist (accessed on 9 September 2023).

32. Lyons, M.; Kamachi, M.; Gyoba, J. The Japanese Female Facial Expression (JAFFE) Dataset. Zenodo . 14 April 1998. Available online: https://doi.org/10.5281/zenodo.3451524 (accessed on 9 September 2023).
33. Parra, D.; Camargo, C. Design Methodology for Single-Channel CNN-Based FER Systems. In Proceedings of the 2023 6th International Conference on Information and Computer Technologies (ICICT), Raleigh, HI, USA, 24–26 March 2023; pp. 89–94. [CrossRef]
34. Angelini, C. Nvidia GeForce GTX 1660 Ti 6GB Review: Turing without the RTX. 2020. Available online: https://www.tomshardware.com/reviews/nvidia-geforce-gtx-1660-ti-turing,6002-4.html (accessed on 11 July 2023).