technologies

MDPI

Article

A TensorFlow Extension Framework for Optimized Generation of Hardware CNN Inference Engines

Vasileios Leon *, Spyridon Mouselinos, Konstantina Koliogeorgi, Sotirios Xydis, Dimitrios Soudris and Kiamal Pekmestzi

School of Electrical & Computer Engineering, National Technical University of Athens, Athens 15780, Greece; mouselinos.spur.kw@gmail.com (S.M.); konstantina@microlab.ntua.gr (K.K.); sxydis@microlab.ntua.gr (S.X.); dsoudris@microlab.ntua.gr (D.S.); pekmes@microlab.ntua.gr (K.P.)

* Correspondence: vleon@microlab.ntua.gr

Received: 16 December 2019; Accepted: 9 January 2020; Published: 13 January 2020

Abstract: The workloads of Convolutional Neural Networks (CNNs) exhibit a streaming nature that makes them attractive for reconfigurable architectures such as the Field-Programmable Gate Arrays (FPGAs), while their increased need for low-power and speed has established Application-Specific Integrated Circuit (ASIC)-based accelerators as alternative efficient solutions. During the last five years, the development of Hardware Description Language (HDL)-based CNN accelerators, either for FPGA or ASIC, has seen huge academic interest due to their high-performance and room for optimizations. Towards this direction, we propose a library-based framework, which extends TensorFlow, the well-established machine learning framework, and automatically generates high-throughput CNN inference engines for FPGAs and ASICs. The framework allows software developers to exploit the benefits of FPGA/ASIC acceleration without requiring any expertise on HDL development and low-level design. Moreover, it provides a set of optimization knobs concerning the model architecture and the inference engine generation, allowing the developer to tune the accelerator according to the requirements of the respective use case. Our framework is evaluated by optimizing the LeNet CNN model on the MNIST dataset, and implementing FPGA- and ASIC-based accelerators using the generated inference engine. The optimal FPGA-based accelerator on Zynq-7000 delivers 93% less memory footprint and 54% less Look-Up Table (LUT) utilization, and up to 10× speedup on the inference execution vs. different Graphics Processing Unit (GPU) and Central Processing Unit (CPU) implementations of the same model, in exchange for a negligible accuracy loss, i.e., 0.89%. For the same accuracy drop, the 45 nm standard-cell-based ASIC accelerator provides an implementation which operates at 520 MHz and occupies an area of 0.059 mm^2, while the power consumption is ∼7.5 mW.

Keywords: machine learning; convolutional neural networks; model optimizations; weight quantization; inference engine optimizations; dataflow optimizations; tensorflow; FPGA; ASIC

1. Introduction

In recent years, significant research has been conducted on the development and optimization of Convolutional Neural Networks (CNNs), a class of Deep Neural Networks (DNNs) that excels in computer vision tasks, such as object recognition [1] and classification [2]. However, such applications belong to the Internet-of-Things (IoT) and high-end embedded system domains, meaning that speed together with energy-efficiency and scalability are of high importance, while general-purpose processors (i.e., CPUs, low-end GPUs) tend to become unfit for these tasks. FPGA-based accelerators [3–6] have started to take their place as viable and promising solutions, thus, investing in their efficient implementation forms an emerging highly valuable design paradigm. ASIC implementations also provide significant gains in performance for DNNs [7–9], even though they lack of reconfigurability.

However, due to the complexity and development overhead of the Hardware Description Language (HDL) approach, the acceleration of CNNs has been extensively studied from the perspective of High Level Synthesis (HLS) design. The authors in [10–12] propose frameworks that allow the user to customize the design of a CNN based on a C high description and attempting to bridge the gap between software and hardware design skills. The aforementioned frameworks are based heavily on utilizing HLS CNN templates, tools and knobs, thus, they introduce performance overheads due to higher behavioral design abstraction, and programmability overheads due to the need for deeply understanding the architectural details. In this paper, the proposed framework surpasses the former inefficiencies, as we provide an HDL library-based, aka highly optimized, approach for the automatic generation of CNN inference engines, directly exposed to the TensorFlow programmer, thus, there is no need for low-level design details.

In the current paper, we expand TF2FPGA [13], a TensorFlow-based framework offering various optimization knobs and techniques to generate an FPGA-based CNN inference engine. More specifically, we extend its capabilities in order to support a more user-friendly interaction by providing a clearer and more expressive API to tune the accelerator. As a result, the interaction is now performed by simply modifying the JSON file of the respective optimization/technique, contrary to [13], where all the available optimizations and techniques are applied via manual work requiring basic HDL knowledge. Overall, the current version of the framework allows software developers to create hardware-based CNN accelerators without any expertise on the HDL development. Moreover, the proposed expanded framework now supports standard-cell-based back-end for ASIC accelerators, thus, the generated HDL inference engine does not target only FPGA-based accelerators as in [13].

The proposed framework allows software developers to reap the benefits of FPGA/ASIC acceleration for their complex CNN models, while avoiding the learning curve of FPGA/ASIC design and the time overhead of HDL development. Hardware and architectural optimizations as well as available optimization techniques of HLS tools, are exposed to the software developer as tuning knobs inside a pool of optimization capabilities. The software programmer has the option to enable or disable the proposed optimizations depending on the requirements of the respective system and the use case. The proposed framework enables automatic and transparent generation of high-throughput FPGA- and ASIC-based CNN accelerators, by extending the well known TensorFlow [14,15] system with automatic acceleration capabilities. It uses a Python inference engine generator that accepts as input the CNN model description and its parameters, i.e., image size, kernel filter window size, number of input and output feature maps etc., declared in a JSON file. The software developer can choose out of a variety of optimization techniques including model architecture optimizations, weight quantization techniques, inference engine optimizations, and dataflow optimizations that apply to the generic CNN architecture. These optimizations can be applied incrementally and deliver a final accelerator that fits the requirements w.r.t. the desired accuracy and performance trade-offs. We note that the generated inference engine is a synthesizable VHDL code, ready to be deployed on an FPGA device or synthesized with a standard cell library targeting a semi-custom ASIC.

The framework is validated using a LeNet CNN model [16] as use case, operating on data from the MNIST dataset for handwritten digits recognition [17]. The optimization options and steps of the framework are implemented for this network and result in two hardware accelerators with 93% less memory footprint and 0.89% accuracy loss: (i) an FPGA-based accelerator with 54% less LUT utilization and up to 10× speedup vs different GPU, CPU combinations, and (ii) an ASIC-based accelerator which operates at 520 MHz, occupies 0.059 mm^2 area and consumes ∼7.5 mW power.

Overall, the contributions of this work are summarized as follows:

1. an automatic TensorFlow-based framework, written in Python, which generates optimized FPGA/ASIC-based CNN accelerators.
2. a variety of optimization knobs and techniques concerning the model architecture and the HDL inference engine, allowing the tuning of the accelerator and the exploration of various trade-offs in terms of hardware resources, performance and accuracy.

3. fully transparency in the generation of the hardware CNN inference engine, allowing the software developers to create FPGA/ASIC-based CNN accelerators without any expertise on HDL development.
4. an in-depth analysis and exploration of the framework capabilities using the LeNet model on the MNIST dataset.
5. a strong experimental evaluation of the generated inference engine using industrial tools, i.e, Xilinx Vivado [18] and Synopsys Design Compiler [19], for the the FPGA- and ASIC-based accelerators, respectively.

The remainder of the paper is organized as follows. Section 2 provides an overview of the related work. Section 3 describes the framework and each one of the proposed optimization techniques. Section 4 introduces the utilized use case, describes how the framework was applied to this specific network and provides experimental evaluation of all metrics of interest. Finally, Section 5 concludes the paper.

2. Related Work

An extensive exploration of the merits of leveraging FPGA technology for accelerating CNNs has been conducted and an overview of the most outstanding works has been reported on extended surveys [20]. Works included in these surveys span from optimized HDL CNN implementations to HLS-based designs in the most recent years and even mature CNN-to-FPGA toolflows.

A significant part of the literature accelerates CNNs by focusing on architectural and algorithmic optimizations. Works from academia are built around the optimization knobs available in third-party HLS tools, while leading third-party tool vendors provide dedicated products to support seamless FPGA acceleration of TensorFlow or Caffe CNNs. In [11], an FPGA-based CNN inference accelerator is synthesized from multi-threaded C software using the LegUp HLS tool. The use of HLS permits a range of architectures to be evaluated, through exploration of the HLS provided tuning knobs. The Vitis AI [21] is a specialized development environment for accelerating AI inference on Xilinx embedded platforms and FPGAs. Vitis AI development environment supports the industry's leading deep learning frameworks like TensorFlow and Caffe, and offers comprehensive APIs to optimize and compile trained networks to achieve the highest AI inference performance for the deployed application.

The authors in [12] present LeFlow, an open-source tool-kit that allows software developers to implement deep neural networks on FPGAs without having any hardware development expertise. The kit generates an LLVM IR from Tensorflow and imports it into an FPGA HLS tool after significant manual transformations. All HLS optimizations are provided as tuning capabilities on the python level. The authors in [22] propose a code generation tool that outputs fully synthesizable Verilog. Their tool combines concatenate-and-pad (CaP) and overlap-and-add (OaA) techniques, which significantly reduce the computational complexity of operations in CNN layers, and frequency domain loop tiling techniques in order to generate high throughput CNN accelerators. These works focus on application-specific optimization techniques leveraging the HLS tools built-in optimization capabilities. Therefore, they require an in-depth understanding of the available techniques, as well as with the architectural details of both the target device and the computational kernel. In our work, this level of understanding and background knowledge is not necessary. We introduce a first level of optimization upon creation of the model and target the model architecture through the adoption of a wide-and-shallow version of the deep complex model. Therefore we tackle this way the computational complexity of the model.

The work presented in [23] proposes a systolic array convolution architecture to achieve better frequency and, thus, performance for CNNs on FPGAs. The optimization of the architecture is achieved through an analytical model and a design space exploration scheme that examine mapping of data on a Processing Element (PE) array, PE array shape and data reuse. These features are incorporated into an end-to-end automation flow, that performs CNN design generation from high-level C code to FPGA. The authors in [4] propose a framework that translates the input CNN to a sequence of instructions,

executed by the computation engine. The engine consists of an array of PEs and is independent of the input CNN model. The authors also employ a data quantization strategy that is implemented in a dynamic way across layers and takes place during the training phase. An extension of this work is presented in [24], where the authors propose an end-to-end compiler that integrates optimizers for graphs, loops and data layouts. The main optimization utilized targets fusion of graph parts, operations, layers, operations across different kernels, etc., and exploring effective fusion strategies. Regarding data management, the proposed strategy is to load all the data of kernels first and then perform the convolution with popping out feature data sequentially. Ma et al. [25] present a register transfer level compiler to automatically generate FPGA-based accelerators. The compiler leverages an RTL module library that has been developed for different types of CNN layers and has been optimized based on a strategy for parallel computation, data movement and memory access presented in [26]. A hardware–software co-design framework that leverages the reconfigurability of FPGAs to efficiently implement a sparse CNN is presented in [27]. A data sparsification scheme optimizes the structure of the sparse CNN during training phase based on memory allocation and data access regularization and an architecture composed of multiple processing elements (PEs) implements the computation parallelism and data reuse feature. Authors in [28] propose a spatial architecture based on a new CNN dataflow which works on both convolutional and fully connected layers. The resulting dataflow is optimized in terms of data movement and energy consumption thanks to an analysis framework that can quantify the energy efficiency of different CNN dataflows under the same hardware constraints. In this work, the proposed framework addresses the bottleneck introduced by data buffering in matrix multiplications of the CNN model through a dataflow architecture that customizes the datapath and and creates a pipeline that allows for computation without stalls. Our framework also extends the quantization-aware training, that can be found across literature, and suggests a quick and simple alternative of post-training optimization.

Recent works have also examined the creation of end-to-end tool flows that incorporate advanced compiler techniques and mitigate the overhead of HDL development timeframe. The authors in [10] present a framework for the automation of HLS design of CNNs. This framework allows the user to customize the design of an equivalent CNN, and generates both a synthesizable C++ code and ready-to-use scripts for Xilinx Vivado. In [24] the authors propose an end-to-end compiler that integrates optimizers for graphs, loops and data layouts and aim at generating more smart instructions. The authors in [29] propose a uniformed mathematical representation for efficient FPGA acceleration of all layers in CNN/DNN models and a framework that finds the optimal mapping of this representation to the specialized accelerator based on roofline model. The generated accelerator is part of Caffeine, a hardware/soft-ware co-designed library that efficiently accelerate the entire CNN on FPGAs. An automated flow receives input from frameworks such as Caffe to program the desired CNN and build the FPGA accelerator. Similarly, we also propose an automatic TensorFlow-based framework, written in Python, which generates optimized CNN accelerators. However the accelerators generated from our framework can be either FPGA or ASIC-based ones.

3. Framework for CNN Inference Generation

The proposed framework, which is illustrated in the block diagram of Figure 1, incorporates an end-to-end CNN inference engine generator including modeling, training, and inference optimizations. Interestingly, it is an extension of the well-established TensorFlow framework [14,15], which trains the model and extracts the final model description and weights. The extracted TensorFlow model is parsed by a custom Python script that generates the HDL inference engine using an in-house optimized CNN layer library. Our layer library is written in VHDL and includes the most widely used layers and functions of CNN models, e.g., convolutional layer, pool layer, ReLU function, etc. Their implementation is based on the corresponding TensorFlow descriptions.

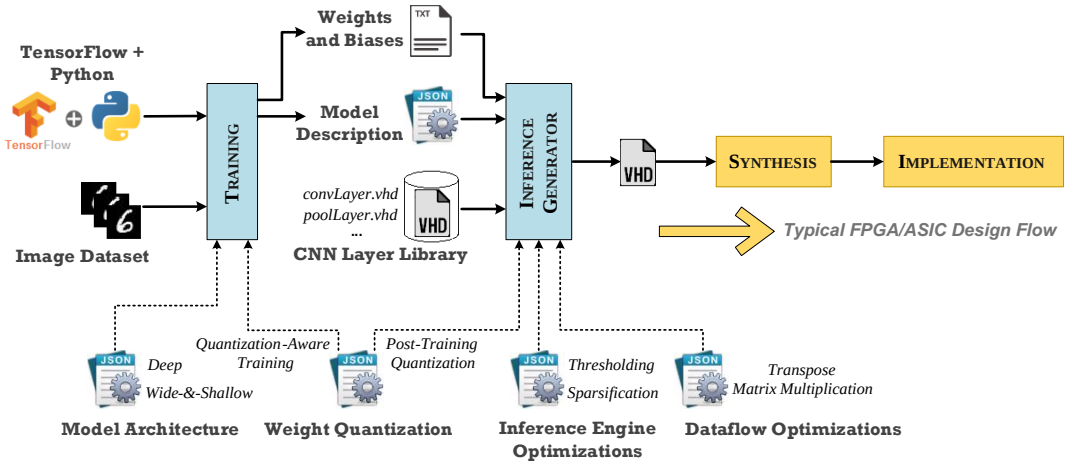


Figure 1. Proposed framework for generating high-throughput CNN hardware accelerators with TensorFlow. The user needs only to modify the JSON files to tune both the training and inference generation, and run the Python scripts (*Training*, *Inference Generator*) to produce a synthesizable HDL inference engine, ready to be deployed on an FPGA or implemented for semi-custom ASIC.

The generated HDL inference engine is synthesizable and can be given as input source to industrial-strong FPGA design tools (e.g., Xilinx Vivado [18], Intel Quartus Prime [30], Synopsys Synplify Pro [31], etc.) and semi-custom ASIC design tools employing standard cell libraries (e.g., Synopsys Design Compiler [19]). Subsequently, the typical FPGA/ASIC development flow is followed to deliver the hardware accelerator. The developer's interaction with our framework is limited only to the configuration of the JSON files and the execution of the Python scripts for training and inference generation. Specifically, the developer can tune both the aforementioned processes by enabling model transformations and optimization knobs in the corresponding JSON files.

3.1. Framework Configuration Capabilities

Regarding the CNN model architecture, the developer can explore the possibility for deep to wide-and-shallow conversion, in order to reduce the execution time and memory footprint. Moreover, two different weight quantization options allow the developer to exploit the fixed-point power in the calculations. Targeting a high-throughput accelerator, we have given the flexibility to the developer to enable optimizations concerning the generation of the engine. These optimizations can be divided into two subcategories. The first category introduces a first level of optimization, and includes application-specific inference engine optimizations, such as weight matrix sparsification or input thresholding. The second category further enhances the performance of the design, and includes dataflow transformations.

The proposed framework allows the programmer to exploit all the options described above, regarding Model Architecture, Weight Quantization, Inference Engine Optimizations and Dataflow Optimizations. Therefore, the programmer can effectively customize the inference engine to fit the specific use-case criteria such as accuracy, timing constraints, etc.

3.2. Model Architecture

DNNs exhibit state-of-the-art performance in terms of accuracy, however, the increased number of model parameters imposes high memory requirements and bottlenecks at the inference execution time due to the huge number of arithmetic operations [32]. Thus, they are not well-suited for applications running on embedded devices with memory limitations and real-time constraints. Towards this direction, significant research has been conducted on the model compression [33,34], producing a wide-and-shallow version of the deep complex model. In general, a deep model is small in node width and large in layer depth, while a wide-and-shallow model is large in node width and shallow in

layer depth. Taking advantage of the reduced memory footprint and simplicity of the wide-and-shallow model's architecture, our framework enables a user option to select whether to convert the deep model to wide-and-shallow or not. The adoption of the wide-and-shallow model offers significant benefits, i.e., the overall training time is heavily reduced and the required memory for the model's weights-and-biases is getting smaller. The real tricky point of such a model is to tune the procedures needed to train the model.

3.3. Weight Quantization

Deep neural networks are increasingly used in applications running on mobile and embedded devices. Consequently, there is an emerging need to reduce the model size in order to comply with the constraints in memory and power dissipation, as well as the communication requirements for transferring the model to the device. Moreover, these devices tend to have lower computing power, especially for arithmetic operations (e.g., floating-point calculations).

As an alternative solution to the floating-point models and architectures, several fixed-point implementations of deep neural networks have appeared, tackling the aforementioned constraints. Significant research has been conducted aiming to convert the model to its more efficient fixed-point equivalent by quantizing the model weights [35], while delivering negligible accuracy drop. Inspired by the promising results of weight quantization [36], we have incorporated two different 8-bit integer quantization techniques to our framework, i.e., a straightforward post-training rounding and a quantization-aware training with TensorFlow Lite [37].

3.3.1. Post-Training Quantization

This framework option is applied to an already-trained float TensorFlow model. More explicitly, it rounds the float weights to the nearest 8-bit integers without re-training the model, offering a quick and simple weight compression and eliminating the need for validation data. A code snippet of our framework handling the post-training quantization is illustrated in Listing 1.

Listing 1. Post-training integer quantization by rounding to the nearest integer.

```
def main():
    with open('quantization.json') as quant_json:
        data = json.load(quant_json)
        for p in data['Weight Quantization']:
            print('Post-Training Quantization:')
            print(p['post-train'])
        if p['post-train'] == "TRUE":
            post_training_quantization()

def post_training_quantization():
    fl = open("float32_weights.txt", "r")
    fx = open("unit8_weights.txt", "w")
    for x in fl:
        fx.write(str(round(float(x))))
        fx.write("\n")
```

3.3.2. Quantization-Aware Training

The quantization-aware training perform the quantization during the training phase, delivering higher accuracy than the post-training technique due to the inherent and iterative error minimization. TensorFlow Lite performs a fake-quantization node insertion and re-trains the CNN graph [37]. In more detail, it models the quantization error using the fake-quantization nodes to simulate the effect of quantization in the forward and backward passes. The forward-pass models quantization, whereas the backward-pass models quantization as a straight-through estimator. Throughout the procedure, TensorFlow Lite tries to minimize the loss function and maintain the original evaluation accuracy of the model. The code snippet demonstrating both training and evaluation is shown in Listing 2.

To produce real integer computations from the trained quantization model, TensorFlow Lite converts the evaluation graph to a fully quantized model, by providing it to the TensorFlow Lite Optimizing Converter (TOCO) as shown in the code snippet of Listing 3.

Listing 2. Quantization-aware training using TensorFlow Lite [37]: fake quantized training and evaluation graph generation.

```
# build Forward Pass of model
loss = tf.losses.get_total_loss()

# rewrite graph w/ fake-quantization nodes for training
g = tf.get_default_graph()
tf.contrib.quantize.create_training_graph(
  input_graph=g,
  quant_delay=QDELAY)

# call Backward Pass optimizer
optimizer = tf.train.GradientDescentOptimizer(learn_rate)
optimizer.minimize(loss)

# build Evaluation model
logits = tf.nn.softmax_cross_entropy_with_logits_v2(...)

# rewrite graph w/ fake-quantization nodes for evaluation
g = tf.get_default_graph()
tf.contrib.quantize.create_eval_graph(input_graph=g)
```

Listing 3. Generation of quantized model using TensorFlow Lite Optimizing Converter (TOCO) [37]: conversion of the frozen evaluation graph to a fully quantized model.

```
toco \
 --input_file = frozen_eval_graph.pb \
 --output_file = tflite_model.tflite \
 --input_format = TENSORFLOW_GRAPHDEF \
 --output_format = TFLITE \
 --inference_type = QUANTIZED_UINT8 \
 --input_shape = "1,224, 224,3" \
 --input_array = inputs \
 --output_array = outputs \
 --std_value = 127.5 \
 --mean_value = 127.5
```

3.4. Inference Engine Optimizations

As a first level of optimization for the HDL inference engine, we employ techniques, such as weight matrix sparsification and input thresholding, to reduce the design complexity and facilitate the arithmetic operations in exchange for accuracy drop. Using sparse weight matrices and converting the operation of convolutional layers to sparse matrix multiplications leads to highly efficient computations [38]. Moreover, a design alternative to perform efficient multiplication, rather than introducing approximations in the circuit [39,40], is to reduce the operator's bit-width. Towards this direction, input thresholding [13] generates a binary image (with 1 s and 0 s), eliminating the need for conventional multipliers in the 1st convolutional layer. The aforementioned optimization are application- and model-specific, and therefore, the developer can assess if they fit to his/her application and CNN model.

3.5. Dataflow Optimizations

The second level of inference optimizations targets the computational kernel of matrix multiplication, that is common for all CNN architectures. Specifically, in order to facilitate the streaming fashion of the data, we transformed the matrix multiplication operation as presented

in [13]. Considering the matrix multiplication $A_{1\times N} \cdot B_{N\times M}$, where A is the input vector and B is the weight matrix, the output vector $C_{1\times M}$ is calculated as follows:

$$c_j = \sum_{i=1}^{N} a_i \cdot b_{ij} \qquad j = 1, 2, ..., M \tag{1}$$

The typical implementation of the matrix multiplication requires input data buffers, first to be filled, and then execute the multiplication. This data buffering imposes inefficient resource utilization regarding the hardwired multipliers, interrupting the continuous operation of a pipelined design. In order to tackle the aforementioned bottleneck, we perform the matrix multiplication by transposing the weight matrix and implementing M parallel FIR filters (MACs), as illustrated in Figure 2. All the FIR filters take as input the same streamed data (a_i). The j-th FIR filter uses the j-th column of the weight matrix and produces c_j. The output vector is partially created, and when the last data (a_N) is served, it is forwarded to the next layer, and thus, the processing of the next batch can start with no delay.

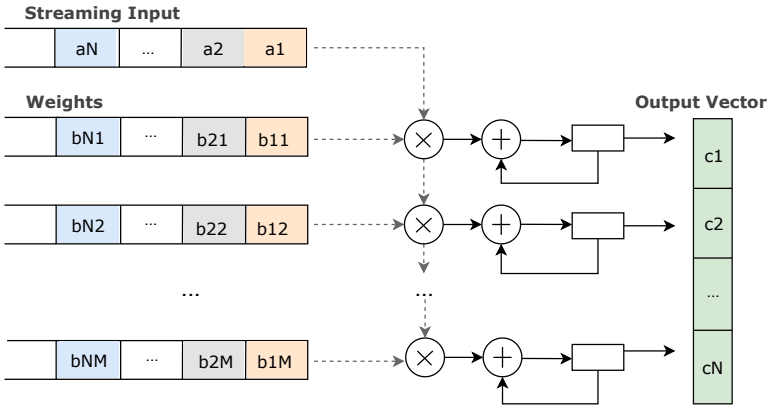


Figure 2. Dataflow optimization [13]: matrix multiplication is performed by implementing parallel FIR filters, facilitating data streaming.

4. Experimental Evaluation Analysis

We showcase the effectiveness of our framework and validate the proposed methodology targeting a specific CNN architecture. Specifically, all examined models are different configurations of the LeNet model [16] and have been trained and tested using the MNIST dataset [17] to perform handwritten digit recognition. The framework for the model development is TensorFlow 1.10, compiled with and without the AVX2 optimized instruction set.

For evaluation purposes, we provide experimental results from the implementation of the generated inference engine on two differing hardware technologies, i.e., the FPGA and the semi-custom ASIC using a standard cell library. We employ industrial-strong tools for both accelerators, i.e., Xilinx Vivado [18] and Synopsys Design Compiler [19].

4.1. Use Case: Optimizing LeNet CNN Model

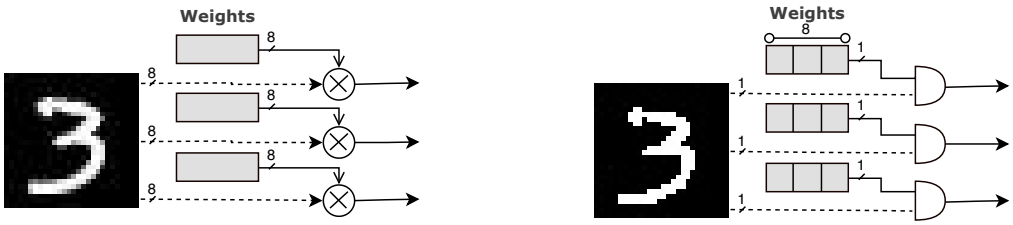
The use case utilized for validation purposes is the LeNet CNN model [16], trained on data from the MNIST [17] dataset in order to perform digit recognition. The tuning knobs and optimization techniques presented in Section 3 are all leveraged in this use case to highlight the effectiveness and flexibility of the proposed framework.

We explore several model architectures considering the accuracy–model complexity trade-off, similarly to the exploration strategy followed in [41]. To ensure that the model preserves the accuracy of the initial LeNet architecture, a careful grid search of sizes, activations and learning rates is conducted. Specifically, we examine the following parameters: (i) number of convolutional layers, (ii) number

of kernels per layer, (iii) size of kernel, (iv) number of fully connected layers, and (v) size of fully connected layer neurons. Throughout the exploration, the models are trained for 100 epochs of batch size 256 and intermediate dropout layer of 65% keep probability. The champion model extracted from the aforementioned exploration consists of a wide-and-shallow (W&S) architecture, one convolutional layer with two 3×3 filters, one 2×2 max pooling layer, and a final 10-neuron fully connected layer. Note that it has also been proven in [33] that a single-layer artificial neural net that has its weights and nodes engineered in order to resemble the behaviour of its deep counterparts on a given task, can perform equally fine in terms of general loss or accuracy.

Concerning weight quantization, we have enabled the quantization-aware training (QAT) technique during the training phase, as it is described in Section 3.3.2. Also, we have explored the impact of the post-training quantization (PTQ) technique described in Section 3.3.1.

Subsequently, we have leveraged the available inference engine optimizations, which are application-specific tuning techniques. The use case of classification of handwritten digits included in the MNIST dataset is a problem of edge-detection. Such problems are proven to be highly robust to noise and artifacts as well as easily captured within 1–2 layers of filtering and non-linearity [42]. In addition, the used images are in single-channeled grayscale 8-bit format, i.e., the pixels are in the range [0–255]. Thus, we exploit these attributes of the problem to propose a pre-processing method that enables further memory reduction with accuracy loss within the use case limits. According to this pre-processing, the pixels under a threshold of 128 are set to 0, and the rest to 1 [13]. The input thresholding (IT) slightly distorts the images, without altering their local information correlations. Thanks to this thresholding technique, in the high-resource consuming convolutional layer, the respective multiplication modules (presented in Figure 3a) can now be replaced by AND gates that use the input pixel as an enable signal (0/1) to forward the ROM stored weights to the next layer, as shown in Figure 3b. As a result, the circuit complexity is significantly decreased, i.e., reduced resource utilization is achieved in the FPGA-based accelerator and less logic cells are employed in the ASIC-based accelerator, and the timing constraints are relaxed. At the same time, the final model's performance is slightly affected (i.e., 0.08% accuracy loss as shown in Table 1).



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

Figure 3. Multiplication in the convolutional layer: (**a**) 8-bit input images require the use of conventional multipliers. (**b**) Input thresholding creates binary images [13], enabling the employment of AND gates rather than multipliers.

As a last level of optimization, we employ the dataflow optimization technique. i.e., the matrix multiplication transformation (MMT). According to this technique, the matrix multiplications of the final fully connected layer are transformed to FIR filters.

Below we examine the impact of the applied techniques and optimizations on both accuracy and weight memory footprint. Subsequently, we present the experimental results for the FPGA- and ASIC-based accelerators.

4.1.1. Accuracy Evaluation

Firstly, we evaluate the accuracy drop of the derived models in comparison with the originally generated models of the MNIST dataset. Both the proposed quantization-aware training and

post-training quantization techniques have been applied to wide-and-shallow converted models. Moreover, we further apply input thresholding to both examined techniques.

As shown in Table 1, the quantization-aware training delivers only 0.81% accuracy loss. On the contrary, the application of the post-training quantization gives poor results, as the accuracy drops over 2%. It is also shown that the input thresholding, if combined with the quantization-aware training, provides extra accuracy drop, i.e., 0.08%, albeit negligible in exchange for the upcoming memory and utilization optimizations.

Table 1. Accuracy loss w.r.t. quantization and thresholding.

Proposed Optimizations	Accuracy Loss (%)
PTQ	2.80
PTQ + IT	3.20
QAT	0.81
QAT + IT	0.89

4.1.2. Memory Footprint Evaluation

Figure 4 presents the memory footprint gains in comparison with the initial deep model. The derived results show that the conversion of the deep model to a wide-and-shallow one results to a 73.1% reduction in the on-board required memory. By further applying the post-training quantization, we achieve a total memory saving of 91%. When selecting quantization-aware training rather than the post-training quantization method, we achieve a total memory saving of 93.4% in comparison with the original model. Considering that quantization-aware training exhibits smaller accuracy loss (see Table 1) as well as slightly larger memory gains (see Figure 4) versus the post-training quantization, we choose to employ it in our final inference engine accelerator.

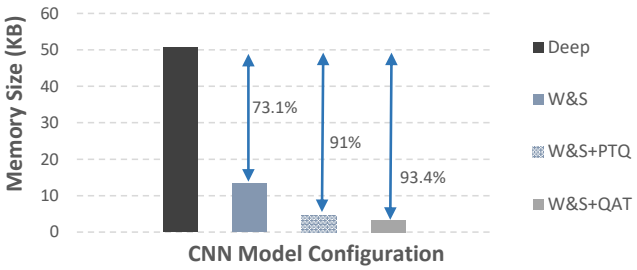


Figure 4. On-board memory footprint for different CNN configurations.

4.2. FPGA-Based Accelerator

Xilinx Vivado has been employed to derive all the designs, with the post-implementation reports of the tool providing the required information to evaluate our work. The target FPGA board is a Zybo Zynq-7000 (XC7Z010) with 28K logic cells, 240KB block RAM and 80 DSP slices. Table 2 presents resource utilization results from the FPGA accelerator. In both implementations, the model has been transformed to wide-and-shallow and the weights are quantized using quantization-aware training. As shown, the weight quantization on its own is not sufficient, as it results in DSP over-utilization (120 out of 80 available DSPs). When employing the proposed input thresholding along with the matrix multiplication transformation, we achieve a 53.6% and 50% reduction in LUTs and DSPs, respectively. This is totally reasonable, as input thresholding eliminates the multipliers and uses AND gates, and thus, LUTs are employed rather than DSPs for the implementation of the convolutional layer. In addition, the matrix multiplication transformation in the fully connected layer slightly reduces the DSP utilization, while it delivers reductions in LUTs, which were originally used for the implementation of the conventional matrix multiplication.

Table 2. FPGA resource utilization w.r.t. proposed optimizations.

Proposed Optimizations	LUT	DFF	DSP
W&S + QAT	11540	2530	120
W&S + QAT + IT + MMT	5353	2536	60

The final FPGA accelerator is based on the wide-and-shallow model and incorporates all the proposed optimizations, i.e., quantization-aware training, input thresholding and matrix multiplication transformation. To assess the accelerator's performance, we perform a comparison vs. CPU/GPU configurations of the same model. The respective CPU/GPU implementations are executed on a 6-core AMD RYZEN 1600 and a 6.1 CUDA capability GPU. The CPU configuration is tested with the AVX2 instruction set both enabled and disabled through the respective compilation flags of TensorFlow environment.

In order to make a fair comparison, we split the execution time according to the software execution phases and isolate the inference execution. Figure 5 presents the results concerning end-to-end as well as per each phase of CPU/GPU execution. The derived results show that the proposed FPGA-based solution outperforms the rest of the configurations, by delivering a 10× speedup at the inference execution vs the fastest software implementation.

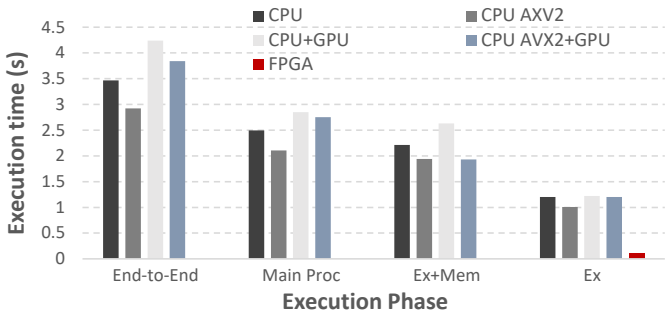


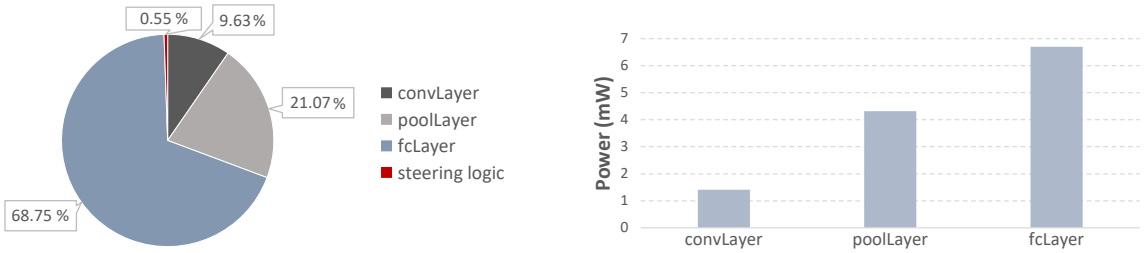
Figure 5. FPGA, CPU and GPU performance for the LeNet inference engine.

Surprisingly, CPU configurations perform better than the GPU ones, mainly due to the limited inter layer parallelism/depth, thus under-utilizing the GPU resources. However, we note that this observation is model-specific, meaning that deeper inference engines would probably be benefited more from GPU execution.

4.3. ASIC-Based Accelerator

The inference engine is synthesized using the TSMC 45-nm standard cell library at the nominal supply voltage (1 V). Similarly to the FPGA-based accelerator, it includes all the applied optimizations (W&S + QAT + IT + MMT). The critical path delay, as reported by the Synopsys Design Compiler tool, is 1.93 ns, while the area is 0.059 mm^2 and the mean power consumption lies around 7.5 mW.

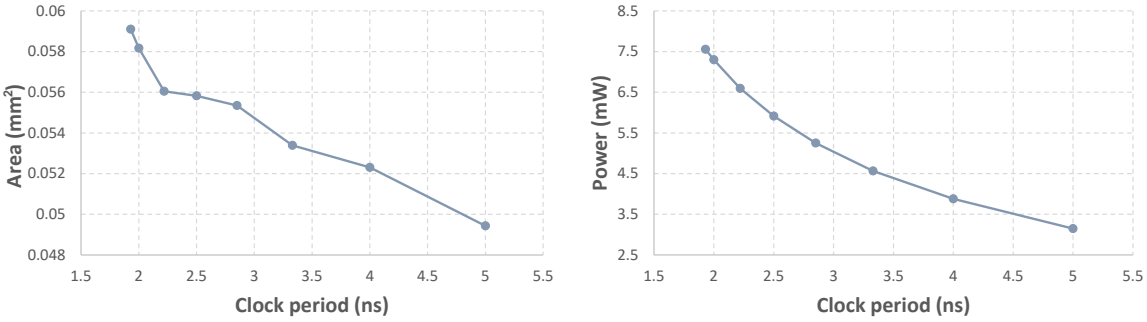
Figure 6 reports the area percentage and the mean power of each network layer, when the inference engine is synthesized at its critical path delay. Regarding area, the convolutional layer, which is optimized by substituting the multiplication with AND gates, consumes less than 10% of the total engine's area. The majority of the area, i.e., ∼70% is occupied by the 10-neuron fully connected layer, which implements the parallel FIR filters for the matrix multiplication. In terms of mean power, the layers consume ∼1–7 mW as standalone components.



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

Figure 6. Standard-cell-based ASIC accelerator of the LeNet inference engine, synthesized at the critical path delay (1.93 ns): area percentage and power consumption per layer. (**a**) Area percentage per layer. (**b**) Mean power per layer.

In Figure 7, we present the variation of area and power w.r.t. the synthesis clock. Starting with a clock of 520 MHz, i.e., the clock that respects the critical path delay, we decrease the frequency to 500 MHz, and then we adopt a constant decrease with a step of 50 MHz. The derived results show that area is decreased almost in a linear fashion, while power decreases exponentially. The area-delay-product (ADP) ranges from 0.11 ns $\times$mm^2 to 0.25 ns $\times$ mm^2, with the high-performance configuration at 1.93 ns providing the best value. On the other hand, the power-delay-product (PDP) remains in the interval 14.6–15.7 ns $\times$ mW. As for ADP, the best PDP is achieved when the inference engine is synthesized at its critical path delay.



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

Figure 7. Standard-cell-based ASIC accelerator of the LeNet inference engine: variation of area and power consumption with respect to the synthesis clock, starting from the critical path delay (1.93 ns). (**a**) Total area; (**b**) Mean total power.

5. Conclusions

In this paper, we propose an HDL library-based framework as an extension to TensorFlow capabilities, which performs automatic generation of CNN FPGA/ASIC-based accelerators and allows the programmer to benefit from the merits of hardware acceleration while surpassing the difficulties of HDL programming. The software programmer has the option to enable or disable the proposed optimizations depending on the requirements of the respective system and the use case. The set of provided optimization techniques includes model architecture optimizations, weight quantization techniques, inference engine optimizations, and dataflow optimizations that apply to the generic CNN architecture. The framework is validated through incremental use of the former optimizations, using a LeNet CNN model [16] as use case and operating on data from the MNIST dataset for handwritten digits recognition [17]. The resultant optimal FPGA-based accelerator exhibits 93% less memory size and 54% LUT utilization reduction, in exchange for 0.89% accuracy loss. In comparison with various GPU, CPU combinations, the accelerator delivers up to 10$\times$ speedup on the inference engine execution. Furthermore, the standard-cell-based ASIC accelerator achieves a critical path delay of 1.93 ns for the same accuracy loss, while the area is 0.059 mm^2 and the mean power values lie around 7.5 mW.

Author Contributions: Conceptualization, S.X. and K.P.; Investigation, V.L.; Methodology, D.S.; Software, V.L. and S.M.; Supervision, S.X. and K.P.; Validation, V.L. and S.M.; Visualization, V.L. and K.K.; Writing—original draft, V.L., S.M. and K.K.; Writing—review & editing, S.X., D.S. and K.P. All authors have read and agreed to the published version of the manuscript

Funding: This research received no external funding.

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

Abbreviations

The following abbreviations are used in this manuscript and refer to the applied CNN optimizations:

W&S	Wide-and-Shallow
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
IT	Input Thresholding
MMT	Matrix Multiplication Transformation

References

1. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2014**, arXiv:1409.1556.
2. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. In Proceedings of the Conference on Neural Information Processing Systems (NIPS), Lake Tahoe, NV, USA, 3–6 December 2012; pp. 1097–1105.
3. Sharma, H.; Park, J.; Mahajan, D.; Amaro, E.; Kim, J.K.; Shao, C.; Mishra, A.; Esmaeilzadeh, H. From High-level Deep Neural Models to FPGAs. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12.
4. Guo, K.; Sui, L.; Qiu, J.; Yao, S.; Han, S.; Wang, Y.; Yang, H. Angel-Eye: A Complete Design Flow for Mapping CNN onto Customized Hardware. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Pittsburgh, PA, USA, 11–13 July 2016; pp. 24–29.
5. Venieris, S.; Bouganis, C. fpgaConvNet: Automated Mapping of Convolutional Neural Networks on FPGAs. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 291–292.
6. Abdelouahab, K.; Pelcat, M.; Sérot, J.; Bourrasset, C.; Berry, F. Tactics to Directly Map CNN Graphs on Embedded FPGAs. *IEEE Embed. Syst. Lett.* **2017**, *9*, 113–116.
7. Conti, F.; Benini, L. A Ultra-low-energy Convolution Engine for Fast Brain-inspired Vision in Multicore Clusters. In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 9–13 March 2015; pp. 683–688.
8. 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Salt Lake City, UT, USA, 1–5 March 2014; pp. 269–284.
9. Qadeer, W.; Hameed, R.; Shacham, O.; Venkatesan, P.; Kozyrakis, C.; Horowitz, M.A. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. In Proceedings of the International Symposium on Computer Architecture (ISCA), Tel Aviv, Israel, 23–27 June 2013; pp. 24–35.
10. Del Sozzo, E.; Solazzo, A.; Miele, A.; Santambrogio, M.D. On the Automation of High Level Synthesis of Convolutional Neural Networks. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 217–224.
11. Kim, J.H.; Grady, B.; Lian, R.; Brothers, J.; Anderson, J.H. FPGA-based CNN Inference Accelerator Synthesized from Multi-threaded C Software. In Proceedings of the IEEE International System-on-Chip Conference (SOCC), Munich, Germany, 5–8 September 2017; pp. 268–273.
12. Noronha, D.H.; Salehpour, B.; Wilton, S.J.E. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. In Proceedings of the International Workshop on FPGAs for Software Programmers (FSP), Dublin, Ireland, 31 August 2018; pp. 1–8.

13. Mouselinos, S.; Leon, V.; Xydis, S.; Soudris, D.; Pekmestzi, K. TF2FPGA: A Framework for Projecting and Accelerating Tensorflow CNNs on FPGA Platforms. In Proceedings of the International Conference on Modern Circuits and Systems Technologies (MOCAST), Thessaloniki, Greece, 13–15 May 2019; pp. 1–4.
14. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A System for Large-scale Machine Learning. *arXiv* **2016**, arXiv:1605.08695.
15. TensorFlow. Available online: http://github.com/tensorflow/tensorflow (accessed on 5 December 2019).
16. Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based Learning Applied to Document Recognition. *Proc. IEEE* **1998**, *86*, 2278–2324.
17. MNIST Database. Available online: http://yann.lecun.com/exdb/mnist/ (accessed on 5 December 2019).
18. Xilinx Vivado. Available online: http://www.xilinx.com/products/design-tools/vivado.html (accessed on 5 December 2019).
19. Synopsys Design Compiler. Available online: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html (accessed on 5 December 2019).
20. Mittal, S. A survey of FPGA-based Accelerators for Convolutional Neural Networks. *Neural Comput. Appl.* **2018**, 1–31, doi:10.1007/s00521-018-3761-1.
21. Xilinx Vitis. Available online: http://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (accessed on 5 December 2019).
22. Zeng, H.; Chen, R.; Zhang, C.; Prasanna, V. A Framework for Generating High Throughput CNN Implementations on FPGAs. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 25–27 February 2018; pp. 117–126.
23. Wei, X.; Yu, C.H.; Zhang, P.; Chen, Y.; Wang, Y.; Hu, H.; Liang, Y.; Cong, J. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In Proceedings of the ACM Design Automation Conference (DAC), Austin, TX, USA, 18–22 June 2017; p. 29.
24. Xing, Y.; Liang, S.; Sui, L.; Jia, X.; Qiu, J.; Liu, X.; Wang, Y.; Shan, Y.; Wang, Y. DNNVM: End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2019**, doi:10.1109/TCAD.2019.2930577.
25. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. An Automatic RTL Compiler for High-throughput FPGA Implementation of Diverse Deep Convolutional Neural Networks. In Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–8.
26. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Vienna, Austria, 3–6 April 2017; pp. 45–54.
27. Li, S.; Wen, W.; Wang, Y.; Han, S.; Chen, Y.; Li, H. An FPGA Design Framework for CNN Sparsification and Acceleration. In Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 28–28.
28. Chen, Y.-H.; Emer, J.; Sze, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In Proceedings of the International Symposium on Computer Architecture (ISCA), Seoul, South, 18–22 June 2016; pp. 367–379.
29. Zhang, C.; Sun, G.; Fang, Z.; Zhou, P.; Pan, P.; Cong, J. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *7*, 2072–2085.
30. Intel Quartus Prime. Available online: http://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html (accessed on 5 December 2019).
31. Synopsys Synplify Pro. Available online: http://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-pro.html (accessed on 5 December 2019).
32. Romero, A.; Ballas, N.; Kahou, S.E.; Chassang, A.; Gatta, C.; Bengio, Y. FitNets: Hints for Thin Deep Nets. *arXiv* **2014**, arXiv:1412.6550.
33. Ba, L.J.; Caurana, R. Do Deep Nets Really Need to be Deep? *arXiv* **2013**, arXiv:1312.6184.
34. Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. *arXiv* **2015**, arXiv:1503.02531.

35. Lin, D.; Talathi, S.; Annapureddy, S. Fixed Point Quantization of Deep Convolutional Networks. In Proceedings of the International Conference on Machine Learning (ICML), New York, NY, USA, 20–22 June 2016; pp. 2849–2858.
36. Krishnamoorthi, R. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. *arXiv* **2018**, arXiv:1806.08342.
37. TensorFlow Lite: Quantization-Aware Training. Available online: http://github.com/tensorflow/tensorflow/tree/r1.13/tensorflow/contrib/quantize (accessed on 5 December 2019).
38. Liu, B.; Wang, M.; Foroosh, H.; Tappen, M.; Pensky, M. Sparse Convolutional Neural Networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015; pp. 806–814.
39. Leon, V.; Zervakis, G.; Soudris, D.; Pekmestzi, K. Approximate Hybrid High Radix Encoding for Energy-Efficient Inexact Multipliers. *IEEE Trans. Very Large Scale Integr. Syst.* **2018**, *26*, 421–430.
40. Leon, V.; Zervakis, G.; Xydis, S.; Soudris, D.; Pekmestzi, K. Walking through the Energy-Error Pareto Frontier of Approximate Multipliers. *IEEE Micro* **2018**, *38*, 40–49.
41. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
42. Chellappa, R.; Fukushima, K.; Katsaggelos, A.K.; Kung, S.Y.; LeCun, Y.; Nasrabadi, N.M.; Poggio, T.A. Guest Editorial Applications of Artificial Neural Networks to Image Processing. *IEEE Trans. Image Process.* **1998**, *7*, 1093–1096.