sensors

MDPI

Article

Lightweight and Energy-Efficient Deep Learning Accelerator for Real-Time Object Detection on Edge Devices

Kyungho Kim [1], Sung-Joon Jang [1], Jonghee Park [1], Eunchong Lee [1] and Sang-Seol Lee [1,*]

Intelligent Image Processing Research Center, Korea Electronics Technology Institute, Seongnam-si 13488, Republic of Korea
* Correspondence: sslee81@keti.re.kr

Abstract: Tiny machine learning (TinyML) has become an emerging field according to the rapid growth in the area of the internet of things (IoT). However, most deep learning algorithms are too complex, require a lot of memory to store data, and consume an enormous amount of energy for calculation/data movement; therefore, the algorithms are not suitable for IoT devices such as various sensors and imaging systems. Furthermore, typical hardware accelerators cannot be embedded in these resource-constrained edge devices, and they are difficult to drive real-time inference processing as well. To perform the real-time processing on these battery-operated devices, deep learning models should be compact and hardware-optimized, and hardware accelerator designs also have to be lightweight and consume extremely low energy. Therefore, we present an optimized network model through model simplification and compression for the hardware to be implemented, and propose a hardware architecture for a lightweight and energy-efficient deep learning accelerator. The experimental results demonstrate that our optimized model successfully performs object detection, and the proposed hardware design achieves 1.25× and 4.27× smaller logic and BRAM size, respectively, and its energy consumption is approximately 10.37× lower than previous similar works with 43.95 fps as a real-time process under an operating frequency of 100 MHz on a Xilinx ZC702 FPGA.

Keywords: tiny machine learning (TinyML); internet of things (IoT); deep learning; hardware accelerator; edge devices; object detection; field-programmable gate arrays (FPGA)

check for updates

Citation: Kim, K.; Jang, S.-J.; Park, J.; Lee, E.; Lee, S.-S. Lightweight and Energy-Efficient Deep Learning Accelerator for Real-Time Object Detection on Edge Devices. *Sensors* **2023**, *23*, 1185. https://doi.org/10.3390/s23031185

Academic Editors: Pedro Melo-Pinto, Duarte Fernandes, Antonio Silva and João L. Monteiro

Received: 15 December 2022
Revised: 17 January 2023
Accepted: 18 January 2023
Published: 20 January 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Deep learning has been popular because of the availability of computing power and the development of big data [1], and various reviews and discussions on deep learning have been extensively conducted in recent years [2–6]. It has been widely applied in many fields such as image recognition [7], object detection [8], autonomous driving [9,10], and robotics [11]. Moreover, deep learning networks have been shown to be successful for these fields [12], and nowadays it has become important even in the field of IoT with the rapid development of IoT devices and network infrastructure [13]. Accordingly, deep learning operation in real time with low energy on these resource-constrained edge devices has emerged as essential work in the era of IoT [14].

However, deep learning models are generally too complex, and they also require considerable amounts of data and their computation [15]. Model complexity in deep learning is a fundamental issue in terms of model framework, model size, optimization process, and data complexity. Most deep learning models have a complex model framework, such as a convolutional neural network (CNN), and their model size is so huge owing to numerous parameters, layers, and filters. In addition, the configuration, such as layer width and filter size, also affects model size. As a result, running a deep learning model requires so much memory to store those numerous parameters and a tremendous amount of intermediate data. Furthermore, high energy consumption is inevitable for computing their calculation and moving so much data from/to memory. Therefore, it is very challenging to

implement the hardware to accelerate these deep learning models on the battery-operated IoT devices.

Thus, TinyML is an important and emerging area for operating machine learning applications on small embedded IoT devices, and hence it has been actively researched recently [16–20]. It aims at designing and developing algorithms and hardware capable of performing inferences on resource-constrained devices at extremely low energy. Accordingly, it takes into account the characteristics of hardware to be operated and tries to optimize the model for the hardware and reduce computational load and memory demand by deploying approximation and compression, like pruning. Moreover, the hardware design has to be implemented as lightweight to be embedded in low-cost resource-constrained devices and energy-efficient so that the deep learning model works smoothly on the battery-operated devices, and low-latency so as to run in real time on IoT edge devices, considering the fast sensory data streams.

In this paper, we present the optimized network model for hardware to be implemented. The proposed optimized model is based on SqueezeNet [21], which is a mobile-oriented network. We perform model reduction and parameter simplification on the backbone model network through model simplification, and integer quantization is adopted for activation and parameters through model compression. Furthermore, a lightweight and energy-efficient hardware architecture is proposed, and an implemented design is able to perform parallel processing between layers and channels deploying a 3D tensor-like processing element (PE) structure. It results in low latency and reduction in energy consumption. Besides, a small amount of on-chip memory is required owing to the proposed on-chip memory management strategy, which makes the design lightweight and low-power. As a result, the experimental results demonstrate that our optimized model successfully performs object detection, and the proposed hardware design achieves 1.25× and 4.27× smaller logic and BRAM size, respectively, and its energy consumption is approximately 10.68× lower than previous related works with 43.95 fps as a real-time process under an operating frequency of 100 MHz on a Xilinx ZC702 FPGA.

The rest of this paper is structured as follows: Section 2 provides some background of lightweight deep learning techniques with our backbone model and its related works. Section 3 presents the proposed model optimization through model simplification and compression for hardware to be implemented. In Section 4, the proposed hardware architecture is presented for the hardware design to be lightweight and energy-efficient. Experimental results for the performance of the proposed model and hardware architecture are shown in Section 5. Lastly, Section 6 discusses the conclusion.

2. Background

Operating deep learning models on edge devices is quite challenging because of their limited resources and computational capabilities. Thus, it is important to make use of lightweight deep learning models that are suitable for execution on resource-constrained devices. Moreover, it is crucial to design lightweight and energy-efficient hardware owing to the battery limitations of low-cost devices. Consequently, hardware/software (HW/SW) co-optimization is critical to deploying deep learning models on these devices, and considerable related research has been aggressively conducted as well [22–24].

Lightweight deep learning techniques are able to be classified into two categories: lightweight deep learning algorithms and transforming existing models into compact/small ones [25,26]. SqueezeNet [21], as a representative edge-device-oriented deep learning model, is a lightweight deep learning algorithm. This category makes the structure of the network model lightweight to reduce computational complexity and the number of parameters by utilizing a residual block or bottleneck block. In addition, converting existing models into compact ones is achieved by knowledge distillation or model compression, such as pruning, quantization/binarization, and the weight-sharing method. These techniques compress the model size and its computation by eliminating redundant parameters, sharing

common values, and reducing data bits. Many studies [27–30] in this field have progressed deploying deep learning models on the edge devices.

The design and implementation of hardware for algorithms, as well as the utilizing of lightweight algorithms, are also crucial for the practical operation on the resource-constrained IoT devices. It should work with low latency, power, and energy with lightweight designs on these devices. Related studies [31–33] on efficient hardware design have been actively conducted to determine the feasibility of running deep learning models on edge hardware. Furthermore, hardware design, even in low-cost devices, should be optimized and customized with optimal architectures. On-chip or external memory and logic are unnecessarily able to be consumed with a general PE configuration that does not consider the characteristics of the deep learning model for deployment.

2.1. SqueezeNet

SqueezeNet [21], shown in Figure 1, is a representative lightweight deep learning model in terms of model size and the number of parameters for hardware with limited resources, particularly memory, and computational capabilities. It preserves its accuracy with fewer parameters than AlexNet [34].

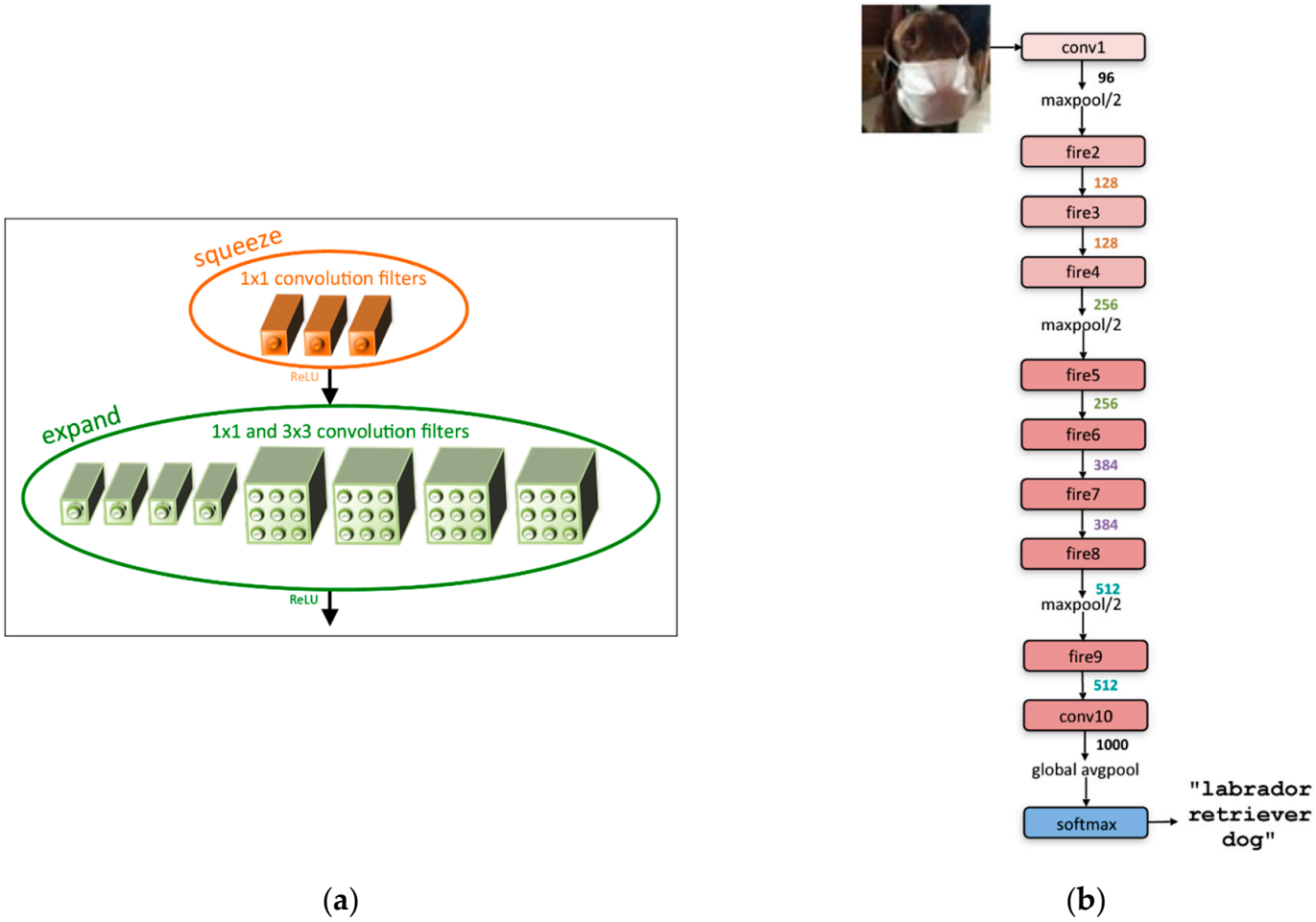


Figure 1. SqueezeNet architecture [21]: (**a**) micro-architectural view: fire module structure in the SqueezeNet; (**b**) macro-architectural view: SqueezeNet comprises a series of fire modules.

A micro-architectural view is shown in Figure 1a. The base unit structure, called the fire module, comprises a squeeze layer with a 1×1 convolution filter and an expand layer that has a mix of 1×1 and 3×3 convolution filters. A ReLU process is performed on the output of the squeeze layer, which feeds into the expand layer, and the final output of the fire module comes from a ReLU operation on the output of the expand layer. Figure 1b shows the macro-architectural view. SqueezeNet consists of a series of fire modules and several maxpool functions between the fire modules.

This model has $9\times$ fewer parameters when replacing the generally used 3×3 filters with 1×1 filters in the squeeze layer. In addition, the base unit fire module makes it possible to reduce the number of channels by deploying a 1×1 convolution filter instead of a 3×3 convolution filter, and the number of channels is expanded again by deploying 3×3 convolution filters in the expand layer. Moreover, the effect of compressed image information can be obtained through sparse down-sampling between a series of fire modules, which reduces the range of the region of interest (RoI) at one glance and leads to higher classification accuracy.

2.2. Related Works

Many studies on accelerating SqueezeNet on FPGA have been introduced in the literature [35–39]. In [35], the authors attempted to enhance the performance of the hardware processing convolution operation through pipelining and loop unrolling and flattening. However, it did not affect the performance owing to the bandwidth bound. In addition, they fused convolution and maxpool operations as layer dimensions, but it had a trivial impact on optimization because their implementation result for resource utilization was too large, and the power dissipation was also too large to deploy the model on edge devices.

Another design and implementation of SqueezeNet, layer-based structured design, was introduced in [36]. The purpose of this design is scalability in constructing CNNs, and it allows the flexible and scalable deployment of the entire CNN. Owing to these characteristics, a large amount of resource utilization was exploited, although a (8–16)-bit fixed quantization strategy was adopted. Moreover, the power consumption was too high, making the design difficult to be embedded and operated on resource-constrained devices.

In [37], they implemented their accelerator with eight multiply-accumulate (MAC) -16 units, which performed 16 MACs in every clock cycle of its operation. In addition, they employed a quantization strategy for parameters such as 8 bits weights, bias, and 16 bits feature maps to reduce their accelerators. In addition, they used various buffers for the parameter and input feature map as an input feature map tile buffer (ITB) and input feature map tile buffer window (ITBW), respectively, avoiding redundant memory accesses that introduce additional power consumption. As a result of their efforts, the power consumption was sufficient to operate on embedded devices, but the memory size was very large, owing to the strategy of utilizing many buffers. Furthermore, the execution time was too long, and therefore an enormous amount of energy was consumed, which made it impracticable to deploy their accelerator on battery-operated edge devices in real time.

A high-speed hardware accelerator was implemented in [38]. The researchers used a ping-pong memory strategy and deployed several first-in, first-outs (FIFOs) in their design to solve the memory bottleneck issue. By preparing a set of twin memories, data from all the series of fire modules can be processed using this ping-pong memory and alternating between a set of twin memories. Additionally, several intermediate FIFOs hold the output data as some pixels of the squeeze layer and pass them to the expand layer when 3×3 window data have been filled. Besides, the authors made use of hardware resources with different configurations in the squeeze and expand layers to speed up layer processing. Consequently, they achieved quite low latency of their hardware accelerator, but their resource utilization on logic and memory was quite high because of the twin memory and FIFOs strategy. Accordingly, it was not suitable for embedding on resource-constrained

edge devices. Furthermore, the power consumption was also high, making it difficult to operate the deep learning model on low-power devices.

To deploy deep learning models for applications such as object detection and image classification on IoT devices, the hardware accelerator embedded on these devices should be lightweight, low-latency, low-power, and low-energy with real-time processing, considering the characteristics of edge devices. To achieve these factors, HW/SW co-optimization is necessarily required in algorithm and hardware.

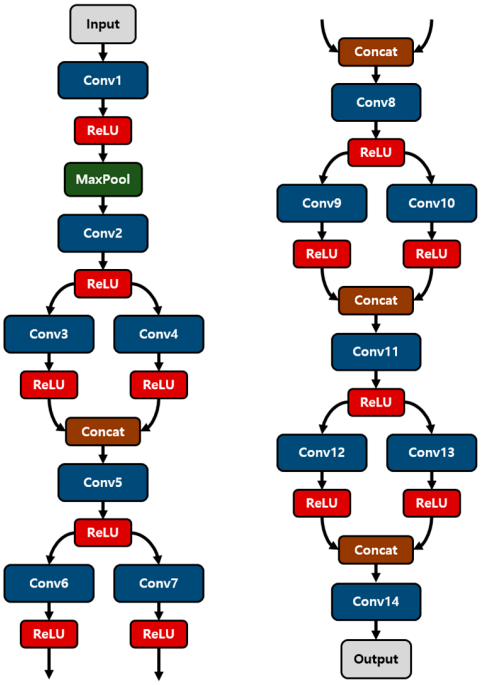
3. Proposed Model Optimization

To obtain well-optimized hardware suitable for deployment on edge devices, transforming deep learning models into compact or small ones is necessary, as well as making use of lightweight deep learning algorithms. Data quantization is a representative method of model compression for operating a model on hardware with limited resources. Additionally, model simplification for optimizing and customizing the hardware to be implemented is crucial for hardware to be more lightweight and energy-efficient. Therefore, we present a model simplification by reducing the model and simplifying the parameter configuration and model compression with data quantization.

3.1. Model Simplification

SqueezeNet, the mobile friendly deep learning model used as our backbone, is a lightweight deep learning algorithm, but it is inevitable that additional optimizations such as model reduction and parameter configuration simplification must be performed, making it executable on low-cost edge devices. In general, the depth of the deep neural network (DNN) is a fundamental issue in terms of the accuracy of the model, the complexity of computation, and runtime. For instance, the deep network shown in Figure 1b involves complex computations with many parameters and intermediate data with high latency and energy consumption, which is not appropriate for low-cost resource-constrained devices. There is a tradeoff between these performances, however; deeper networks or increasing the depth of networks is not always good [40]. Inspired by this, we have conducted model reduction for the model to be lightweight by reducing the depth of the model and involving the intermediate maxpool functions in convolutions with less computation complexity and low latency. Moreover, simplification of the parameter configuration was also performed for the hardware-oriented structure. This is illustrated in Figure 2.

An overview of the simplified model is shown in Figure 2a. It consists of 14 convolutions, 1 maxpool function, and 4 concatenation procedures with several intermediate ReLU processes. One convolution with a ReLU as a squeeze layer and two convolutions with two ReLUs as an expand layer and a final concatenation procedure constitute a fire module; hence, the reduced model has four fire modules in contrast to the eight fire modules of the backbone model in Figure 1b. However, we achieved a successful object detection performance with this reduced model network, as shown in Section 5. In addition, several convolutions have maxpool functions within by adopting stride 2 in the middle of the convolution operation, as shown in Figure 2b. The convolution operations in conv6, conv7, conv9, and con10 in Figure 2b are processed with stride 2 instead of additional maxpool function operations after each convolution process, as shown in Figure 1b. This results in low latency, less computation, and even low energy consumption by involving maxpool functions in the convolution process. In addition, the configuration of the channel number of the filter in Figure 2b indicates a multiple of four, except for the initial primary input, which is three: R, G, and B. This is a constraint for the model to be hardware-oriented, leading to lightweight and energy-efficient hardware with good optimization for resource-constrained devices. The significance of the channel number of the filter to be a multiple of four is described in Section 4.5.1 in detail.



(**a**)

layer name/type	input size				filter size				output size				remarks
	N	C	H	W	N	C	H	W	N	C	H	W	
input image	1	3	192	256	-	-	-	-	1	3	192	256	
conv1	1	3	192	256	16	3	3	3	1	16	96	128	padding, stride2
ReLU	1	16	96	128	-	-	-	-	1	16	96	128	
Maxpool	1	16	96	128	-	-	3	3	1	16	48	64	No padding, stride2
conv2: SqueezeNet1x1	1	16	48	64	4	16	1	1	1	4	48	64	No padding, stride1
ReLU	1	4	48	64	-	-	-	-	1	4	48	64	
conv3: Expand1x1	1	4	48	64	16	4	1	1	1	16	48	64	No padding, stride1
conv4: Expand3x3	1	4	48	64	16	4	3	3	1	16	48	64	padding, stride1
ReLU: conv3	1	16	48	64	-	-	-	-	1	16	48	64	
ReLU: conv4	1	16	48	64	-	-	-	-	1	16	48	64	
Concat	1	16	48	64	-	-	-	-	1	32	48	64	
	1	16	48	64	-	-	-	-					
conv5: SqueezeNet1x1	1	32	48	64	4	32	1	1	1	4	48	64	No padding, stride1
ReLU	1	4	48	64	-	-	-	-	1	4	48	64	
conv6: Expand1x1	1	4	48	64	16	4	1	1	1	16	24	32	No padding, stride2 as a integrated maxpool
conv7: Expand3x3	1	4	48	64	16	4	3	3	1	16	24	32	padding, stride2 as a integrated maxpool
ReLU: conv6	1	16	24	32	-	-	-	-	1	16	24	32	
ReLU: conv7	1	16	24	32	-	-	-	-	1	16	24	32	
Concat	1	16	24	32	-	-	-	-	1	32	24	32	
	1	16	24	32	-	-	-	-					
conv8: SqueezeNet1x1	1	32	24	32	12	32	1	1	1	12	24	32	No padding, stride1
ReLU	1	12	24	32	-	-	-	-	1	12	24	32	
conv9: Expand1x1	1	12	24	32	32	12	1	1	1	32	12	16	No padding, stride2 as a integrated maxpool
conv10: Expand3x3	1	12	24	32	32	12	3	3	1	32	12	16	padding, stride2 as a integrated maxpool
ReLU: conv9	1	32	12	16	-	-	-	-	1	32	12	16	
ReLU: conv10	1	32	12	16	-	-	-	-	1	32	12	16	
Concat	1	32	12	16	-	-	-	-	1	64	12	16	
	1	32	12	16	-	-	-	-					
conv11: SqueezeNet1x1	1	64	12	16	24	64	1	1	1	24	12	16	No padding, stride1
ReLU	1	24	12	16	-	-	-	-	1	24	12	16	
conv12: Expand1x1	1	24	12	16	64	24	1	1	1	64	12	16	No padding, stride1
conv13: Expand3x3	1	24	12	16	64	24	3	3	1	64	12	16	padding, stride1
ReLU: conv12	1	64	12	16	-	-	-	-	1	64	12	16	
ReLU: conv13	1	64	12	16	-	-	-	-	1	64	12	16	
Concat	1	64	12	16	-	-	-	-	1	128	12	16	
	1	64	12	16	-	-	-	-					
conv14	1	128	12	16	135	128	3	3	1	135	12	16	padding, stride1

(**b**)

Figure 2. Model simplification of SqueezeNet to be more compact and smaller, being suitable for execution on the devices having limited resources: (**a**) overview of the model; (**b**) configuration of the model in detail.

3.2. Model Compression

Data quantization, a representative model compression technique, is necessarily required to compress a model on resource-limited, low-latency, and low-energy devices owing to the latter's constraints on compactness and battery capability. Corresponding to the limited memory, computation, and power of these devices, data moving from/to memory and to be processed should be quantized from floating point numbers for small size and low computation and power. The quantization procedure is described in Equation (1):

$$\begin{aligned} Data_q &= Data_f * S_D \\ &= \left(\sum Act_f W_f + B_f\right) * S_D \\ &= \left(\sum (Act_q / S_A)(W_q / S_W) + (B_q / S_B)\right) * S_D \\ &= (S_D / S_A S_W) * \left(\sum Act_q * W_q + \left(\frac{S_A S_W}{S_B}\right) B_q\right) \end{aligned} \quad (1)$$

ording to the comment.cronyms.ai and about abbreviations and acronyms.

This is a convolution operation utilizing symmetric quantization. *Data* comprises activation, weight, and bias components denoted as Act, W, and B, respectively. S indicates a scale factor, and D, f, and q denote data, floating point, and quantization, respectively. Floating point elements are able to be classified into quantized elements and their respective scale factors. For instance, Act_f, floating point activation, is classified into Act_q, quantized activation, and S_A, scale factor of activation. Additional calculations using the zero point in this equation are required for asymmetric quantization. The asymmetric quantization typically has a higher resolution than the symmetric quantization. Thus, the weight parameters were quantized by the symmetric quantization, and the asymmetric technique was adopted for activation quantization.

Data quantization for the convolution operation is shown in Figure 3. It describes the quantized bit of each component with configurations. The weight parameter was quantized to integer 8 bits, and bias, multi-scale, and shift-scale factors were quantized to integer 32, 12, and 8 bits, respectively, in the order of BQ, that is, bias and quantization, as shown in Figure 3. In addition, the activation component was also quantized as 8 bits. These quantized parameters are able to be obtained offline so that employing these quantized elements is sufficient to operate inference process for deploying the deep learning model on devices. Consequently, memory size and computational complexity can be reduced, leading to the feasibility of model deployment on low-cost edge devices.

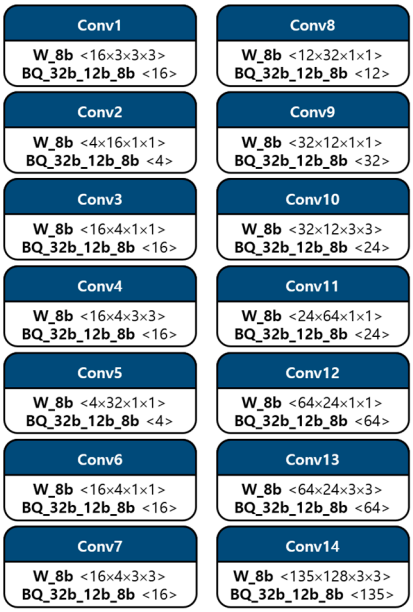


Figure 3. Model compression through data quantization on convolutions.

4. Proposed Hardware Architecture

The design and implementation of hardware are very important for deep learning models to be deployed on resource-constrained edge devices, even if the model has a lightweight algorithm that adopts model compression and simplification. In other words, it is difficult to deploy a lightweight model on such devices when the implemented hardware is bulky with redundant logic and memory requirements and does not have a well-optimized architecture, which leads to the huge size of the hardware resource, high latency, power, and energy consumption. Therefore, we propose a hardware architecture with an optimal design for the presented lightweight deep learning model by utilizing parallel processing between layers and channels through a 3D tensor-like PE structure, and a memory-efficient on-chip memory management strategy.

An overview of the proposed hardware architecture is shown in Figure 4. We performed a customized direct memory access (DMA) design that interacts with an external memory through advanced extensible interface 4 (AXI4) protocol. The inform layer unit controls the order of the layer, and the ifmap (input feature map) driver unit fetches the input feature map data from an external memory through DMA and conducts read/write operations from/to on-chip memories for the ifmap, feeding into an arithmetic core unit. In addition, parameter data such as weights, bias, multi-scale factors, and shift-scale factors are fetched by the parameter driver unit, which reads and writes the fetched parameter data from/to on-chip memories for the parameters. The arithmetic core unit performs convolution and maxpool operations with ifmap data from the ifmap driver unit and parameter data from the parameter driver unit, and the result of the arithmetic core unit is fed into the ofmap (output feature map) driver unit. Finally, the ofmap driver unit performs read/write operations of data output from the arithmetic core unit from/to the on-chip memories for ofmap. The detailed operation of each unit is as follows.

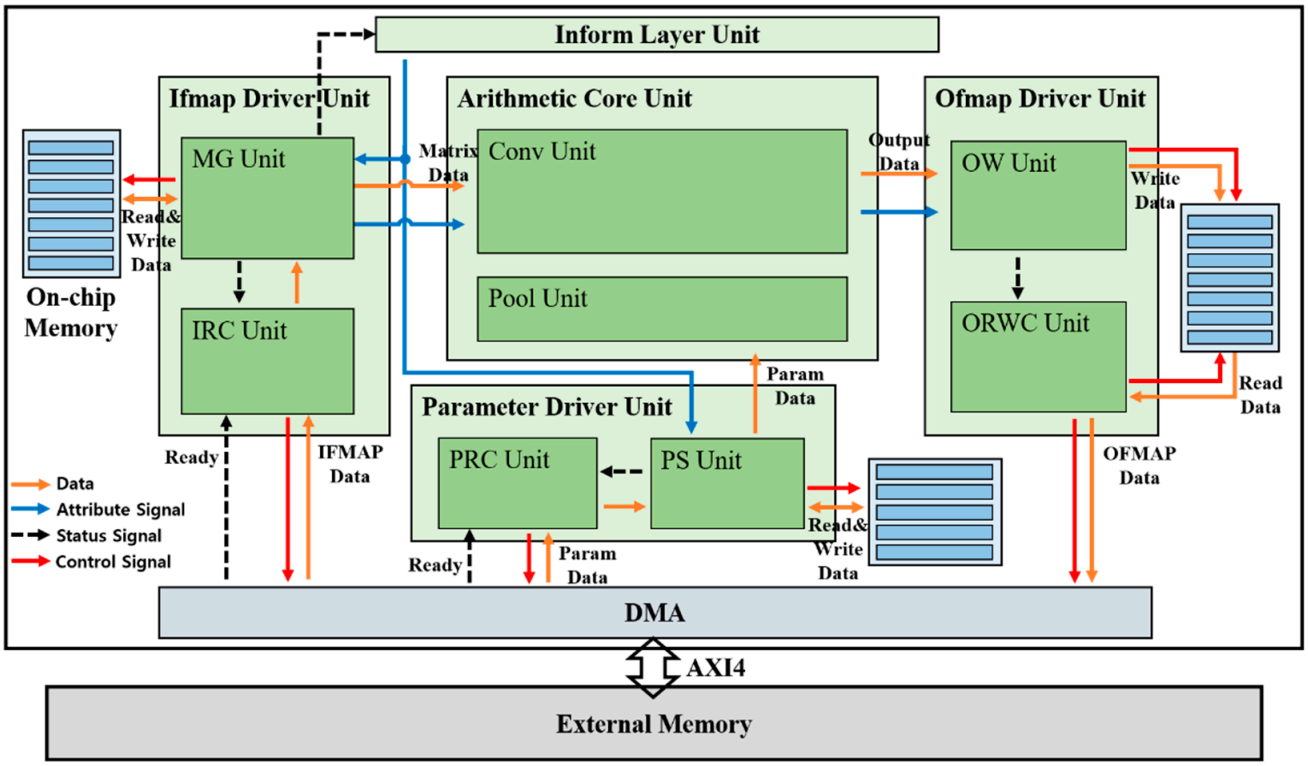


Figure 4. Proposed hardware architecture overview.

4.1. Inform Layer Unit

The layer order is controlled by the inform layer unit, considering the status signal from the ifmap driver unit. The status signal includes information if the loading ifmap data in the current layer has been finished, indicating the availability of the next layer. As a result, the ifmap driver unit and parameter driver unit can be synchronized using the attribute signal as the output signal of the inform layer unit.

4.2. Ifmap Driver Unit

4.2.1. Ifmap Read Control (IRC) Unit

The ifmap read control (IRC) unit fetches ifmap data line by line from an external memory through DMA with a ready signal. The IRC unit passes the loaded ifmap line data to the matrix generation (MG) unit, as described in Section 4.2.2, considering the status signal from the MG unit. Accordingly, this unit enables the MG unit to transform ifmap data to matrix-type.

4.2.2. Matrix Generation (MG) Unit

The MG unit transforms ifmap data to matrix-type for parallel processing between layers, leading to a reduction in latency. In other words, this unit makes it possible to process 3×3 and 1×1 convolutions in an expand layer simultaneously. The matrix generation process in the MG unit is illustrated in Figure 5. The ifmap stream comes to the MG unit line by line from top to bottom of the input feature map in the order of blue, grey, orange, and green. The first line of the ifmap, the blue one in Figure 5, is stored in on-chip memory 0, while on-chip memories 1 and 2 are idle because they are waiting for the next two lines of ifmap, grey and orange in Figure 5. When the second line, the grey one in Figure 5, streams, the first 3×3 matrix output, the dark blue one in Figure 5, comes out with the read state of on-chip memory 0 for the blue ifmap and the read/write state of on-chip memory 1 for the grey ifmap in the case of zero padding on the top line of the 3×3 matrix, which finally consists of zeros on top and blue ifmap data in the middle, and grey ifmap data on the bottom. The second 3×3 matrix output, the dark grey one in Figure 5, is generated when the third ifmap, the orange one in Figure 5, stream comes with read/write operations on on-chip memory 2. This second 3×3 matrix consists of blue ifmap data on top and grey ifmap data in the middle, and orange ifmap on the bottom. Finally, the third 3×3 matrix output, the dark brown one in Figure 5, comes out with overwrite and read operations of the fourth ifmap line, the green one in Figure 5, on on-chip memory 0. In this third 3×3 matrix, grey, orange, and green ifmap data are located on top, middle, and bottom, respectively. Accordingly, the generated 3×3 ifmap matrix includes ifmap data for both 3×3 and 1×1 convolutions at the center of the matrix.

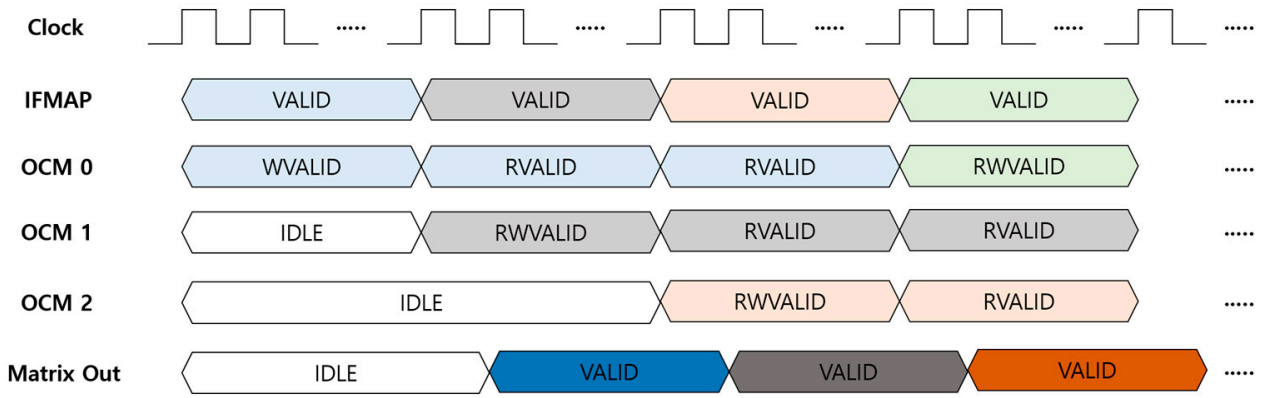


Figure 5. Matrix generation process in MG unit.

By this parallel processing between layers with 3×3 matrix generation, external memory access can be reduced by half compared to the process in the order of 1×1 and 3×3 convolutions in an expand layer because the external memory access needs double for feeding the inputs to the 1×1 and 3×3 convolutions, respectively, at different times. However, this external memory access is the representative issue degrading the performance of the system because it relatively requires so much time and power. Thus, this 3×3 ifmap matrix generation enables the embedded hardware to operate in real time with low latency and low energy. In addition, it can also reduce the space for external memory and on-chip memory. If the two layers in an expand layer are processed in sequence, the ifmap as an input of the expand layer has to be stored in memory by the end of the 1×1 convolution process because the 3×3 convolution should take the same input, resulting in a redundant use of reusable memory. Therefore, redundant memory use or occupancy can be eliminated by the MG unit.

4.3. Parameter Driver Unit

4.3.1. Parameter Read Control (PRC) Unit

The parameter read control (PRC) unit fetches parameter data, such as weights, bias, mult-scale, and shift-scale, from an external memory through DMA with a ready signal. The PRC unit passes the loaded parameter data to the parameter set (PS) unit, as described in Section 4.3.2, considering the status signal from the PS unit. Accordingly, this unit helps the PS unit set the parameter data on time.

4.3.2. Parameter Set (PS) Unit

Parameter data are set in advance as registers for each layer by the PS unit, fed into an arithmetic core unit. The PS unit is synchronized with the MG unit for layer order by the inform layer unit, and the data of each parameter to the arithmetic core unit are also fed at the same time as the ifmap matrix data from the MG unit to the arithmetic core unit. In addition, this PS unit performs a read operation to load parameter data from an external memory through the DMA and a write operation on its own on-chip memories, leading to less external memory access.

4.4. Arithmetic Core Unit

4.4.1. Conv Unit

All convolution operations are conducted in a conv unit. The synchronized ifmap matrix data from the MG unit and quantized parameter data from the PS unit are fed into 3D tensor-like PEs, as shown in Figure 6, indicating a 3×3 convolution case. It has channel (C), height (H), and width (W) components, as CxHxW type, and comprises 3×3 PEs with four numbers of channels in detail, as mentioned in Section 3.1. These 3D tensor-like PEs enable parallel processing between channels, leading to low-latency hardware, and memory-efficient architectures, as discussed in Section 4.5.1. Moreover, the detailed operation on convolution with the quantized data, inside one PE in 3D tensor-like PEs, is shown in Figure 7. Unsigned 8-bit input activation, signed 8-bit weight parameter, signed 32-bit bias parameter, unsigned 12-bit multi-scale factor, and unsigned 8-bit shift-scale factor enter the PE, having multipliers, adders, accumulator, and shifter with a clamping operation.

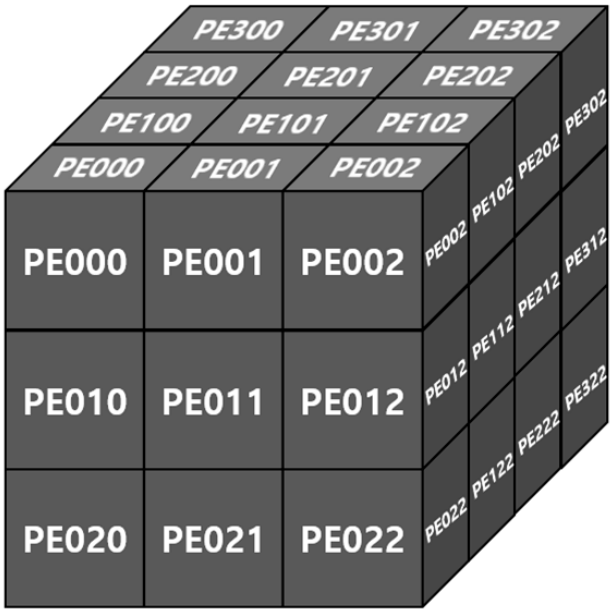


Figure 6. 3D tensor-like PE architecture: 3 × 3 case.

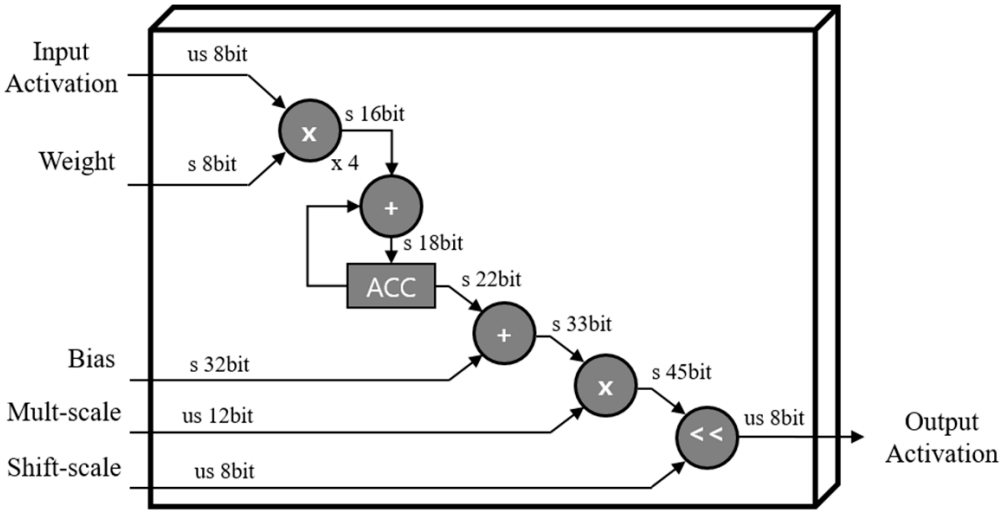


Figure 7. Convolution operation in PE: 1 × 1 case.

4.4.2. Pool Unit

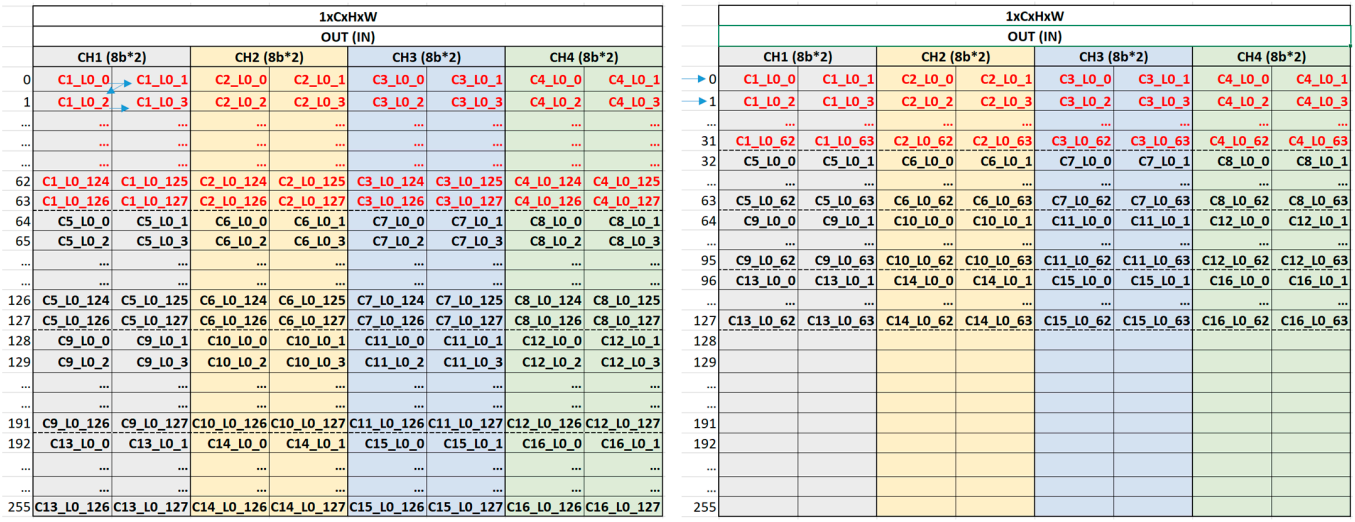
The pool unit operates as a maxpool function in the sequence of layers. It takes 3 × 3 ifmap matrix data and performs maxpooling among 9 ifmap data within a 3 × 3 matrix. The output comes into the ofmap write (OW) unit, as described in Section 4.5.1, in the ofmap driver unit in the maxpool layer.

4.5. Ofmap Driver Unit

4.5.1. Ofmap Write (OW) Unit

The output feature map data from the arithmetic core unit are fed into the OW unit, storing this output feature map on its own on-chip memory, as shown in Figure 8. Figure 8a,b show the on-chip memory status of the OW unit at conv1 and maxpool layer, respectively. The physical on-chip memory has a 256 × 64 configuration as height (address) × width (bit). As for C1_L0_0, C1 and L0 stand for the first channel and line of output feature map, respectively, and the last number 0 indicates the first output feature map data. In other words, C1_L0_0 indicates the first output feature map data in the first line

of the output feature map at the first channel, which is generated one by one, in the order of channel number from C1 to C16 and the order of output feature map width direction from 0 to 127 within a channel, by the arithmetic core unit in the convolution operation, as shown in Figure 8a. Similarly, four output feature map data, C1_L0_0, C2_L0_0, C3_L0_0, and C4_L0_0, are generated at the same time in a maxpool operation as shown in Figure 8b. By gathering the next output feature map data, the data packet {C1_L0_0, C1_L0_1} can be written in the on-chip memory at the conv1 layer and {C1_L0_0, C1_L0_1, C2_L0_0, C2_L0_1, C3_L0_0, C3_L0_1, C4_L0_0, C4_L0_1} can be stored in the on-chip memory at the maxpool layer.



	1xCxHxW OUT (IN)							
	CH1 (8b*2)		CH2 (8b*2)		CH3 (8b*2)		CH4 (8b*2)	
0	C1_L0_0	C1_L0_1	C2_L0_0	C2_L0_1	C3_L0_0	C3_L0_1	C4_L0_0	C4_L0_1
1	C1_L0_2	C1_L0_3	C2_L0_2	C2_L0_3	C3_L0_2	C3_L0_3	C4_L0_2	C4_L0_3
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
62	C1_L0_124	C1_L0_125	C2_L0_124	C2_L0_125	C3_L0_124	C3_L0_125	C4_L0_124	C4_L0_125
63	C1_L0_126	C1_L0_127	C2_L0_126	C2_L0_127	C3_L0_126	C3_L0_127	C4_L0_126	C4_L0_127
64	C5_L0_0	C5_L0_1	C6_L0_0	C6_L0_1	C7_L0_0	C7_L0_1	C8_L0_0	C8_L0_1
65	C5_L0_2	C5_L0_3	C6_L0_2	C6_L0_3	C7_L0_2	C7_L0_3	C8_L0_2	C8_L0_3
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
126	C5_L0_124	C5_L0_125	C6_L0_124	C6_L0_125	C7_L0_124	C7_L0_125	C8_L0_124	C8_L0_125
127	C5_L0_126	C5_L0_127	C6_L0_126	C6_L0_127	C7_L0_126	C7_L0_127	C8_L0_126	C8_L0_127
128	C9_L0_0	C9_L0_1	C10_L0_0	C10_L0_1	C11_L0_0	C11_L0_1	C12_L0_0	C12_L0_1
129	C9_L0_2	C9_L0_3	C10_L0_2	C10_L0_3	C11_L0_2	C11_L0_3	C12_L0_2	C12_L0_3
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
191	C9_L0_126	C9_L0_127	C10_L0_126	C10_L0_127	C11_L0_126	C11_L0_127	C12_L0_126	C12_L0_127
192	C13_L0_0	C13_L0_1	C14_L0_0	C14_L0_1	C15_L0_0	C15_L0_1	C16_L0_0	C16_L0_1
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
255	C13_L0_126	C13_L0_127	C14_L0_126	C14_L0_127	C15_L0_126	C15_L0_127	C16_L0_126	C16_L0_127

(**a**)

	1xCxHxW OUT (IN)							
	CH1 (8b*2)		CH2 (8b*2)		CH3 (8b*2)		CH4 (8b*2)	
0	C1_L0_0	C1_L0_1	C2_L0_0	C2_L0_1	C3_L0_0	C3_L0_1	C4_L0_0	C4_L0_1
1	C1_L0_2	C1_L0_3	C2_L0_2	C2_L0_3	C3_L0_2	C3_L0_3	C4_L0_2	C4_L0_3
...	...	...	...	...	...	...	...	...
31	C1_L0_62	C1_L0_63	C2_L0_62	C2_L0_63	C3_L0_62	C3_L0_63	C4_L0_62	C4_L0_63
32	C5_L0_0	C5_L0_1	C6_L0_0	C6_L0_1	C7_L0_0	C7_L0_1	C8_L0_0	C8_L0_1
...	...	...	...	...	...	...	...	...
63	C5_L0_62	C5_L0_63	C6_L0_62	C6_L0_63	C7_L0_62	C7_L0_63	C8_L0_62	C8_L0_63
64	C9_L0_0	C9_L0_1	C10_L0_0	C10_L0_1	C11_L0_0	C11_L0_1	C12_L0_0	C12_L0_1
...	...	...	...	...	...	...	...	...
95	C9_L0_62	C9_L0_63	C10_L0_62	C10_L0_63	C11_L0_62	C11_L0_63	C12_L0_62	C12_L0_63
96	C13_L0_0	C13_L0_1	C14_L0_0	C14_L0_1	C15_L0_0	C15_L0_1	C16_L0_0	C16_L0_1
...	...	...	...	...	...	...	...	...
127	C13_L0_62	C13_L0_63	C14_L0_62	C14_L0_63	C15_L0_62	C15_L0_63	C16_L0_62	C16_L0_63
128								
129								
...								
...								
191								
192								
...								
...								
255								

(**b**)

Figure 8. On-chip memory management strategy: (**a**) on-chip memory of OW unit at conv1 layer and its memory write operation procedure; (**b**) on-chip memory of OW unit at maxpool layer and its memory write operation procedure.

As shown in Figure 8, the entire data in all channels for one line of the feature map are stored in the on-chip memory and split into four channel sections. This indicates that the arithmetic core unit is able to generate the output feature map data without any bottleneck by fetching the entire data of the input feature map in one line across all channels in the next layer because the entire data across all channels are needed to generate the output feature map data. If the entire data are not able to be provided continuously across all channels, several on-chip memories are additionally required to store the partial sum result. This sequential provision across all channels can be obtained by fetching the data in the order of on-chip memory address owing to the split four-channel section. If the channel section is split into more pieces, for example, eight channel sections, lots of data can be loaded at the same time, but that reduces the flexibility and performance of an algorithm. Similarly, if the channel section is split into fewer pieces, for example, two channel sections, the flexibility of an algorithm can be improved, but some data can be fetched at the same time, leading to the low latency of the hardware. Therefore, the significance of the channel number of the filter to be a multiple of four is on HW/SW co-optimization.

As a result, this configuration of data storage on the on-chip memory of the output feature map is maintained in an external memory so that the input feature map with the same data configuration can be loaded on the on-chip memory of the input feature map. The OUT (IN) on top of the on-chip memory in Figure 8 indicates that the configuration of the output data stored in memory in the current layer is maintained at the input feature map fetch in the next layer. This on-chip memory management strategy enables incremental external memory access, thereby reducing the latency and power consumption caused by

frequent and irregular external memory access. As a result, low latency and low energy consumption can be achieved with a small amount of on-chip memory, enabling the implemented hardware to be more lightweight and suitable for the resource-constrained and battery-operated edge devices.

Figure 9 shows the overall process flow of the implemented hardware, which is related to the data read/write through on-chip memories as described in Figure 8. As shown in Figure 9a, the configuration is 16 × 96 × 128 as channel × height × width, and the first line of the output feature map in the first channel is written first in the order of output feature map width direction from 0 to 127 within a channel in the on-chip memory. Next, the first line of the output feature map in the second channel is written in the same order as the first channel in the on-chip memory. As a result, the entire data in the first line of the output feature map across all channels are written in 16 iterations in the conv1 layer. Moreover, this output feature map data is the same as the input feature map data at the same time in Figure 9b, which indicates the ifmap read operation at maxpool layer as the next layer of conv1. Therefore, the input feature map data across four channels can be loaded incrementally without irregular memory access at the maxpool layer, and the input feature map across all 16 channels is able to be fetched in four iterations, as shown in Figure 9b. In addition, these data as matrix-type are fed into the 3D tensor-like PEs in the arithmetic core unit.

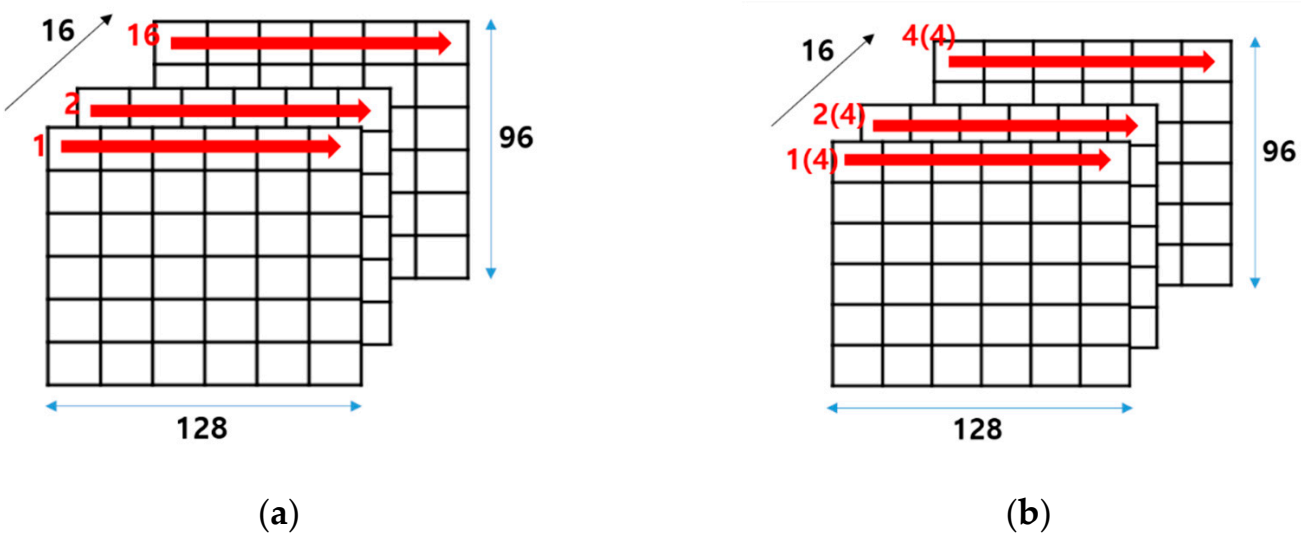


Figure 9. Overall process flow: (**a**) in ofmap write operation on on-chip memory at conv1 layer; (**b**) in ifmap read operation on on-chip memory at maxpool layer, and maxpool operation on maxpool unit.

4.5.2. Ofmap Read/Write Control (ORWC) Unit

Output feature map data written in their on-chip memories are loaded by the ofmap read/write control (ORWC) unit, which writes the loaded data in an external memory through DMA. A pair of on-chip memories for the output feature map data prevents the data from being overwritten before being loaded by the ORWC unit. In other words, the ORWC unit starts to load the written output feature map data when the write operation of the output feature map data in the OW unit is completed, and then the OW unit performs a write operation on another on-chip memory, while the ORWC unit loads the written data from the first on-chip memory. This makes the hardware operate with low latency, and without any bottleneck on resource-constrained edge devices because the on-chip memory is small owing to the on-chip memory management strategy.

5. Experimental Results

Qualitative and quantitative evaluations of the model presented in Section 3 were performed, which indicated that the model successfully carried out object detection regardless of the shape, number, color, and angle of the objects. Moreover, the hardware with the proposed architecture in Section 4 has been implemented in FPGA with less resource utilization and energy consumption compared to related studies, which indicates that the implemented hardware accelerator enables the presented model to be operated with low resource use and energy consumption in real time on the resource-constrained and battery-operated edge devices.

5.1. Performance of Proposed Model

The presented model conducted object detection on a dataset with 13,041 GT, as shown in Table 1. It manifests a recall performance of 93.1% with true positive (TP) and false negative (FN) components, and a precision of 82.6% with TP and false positive (FP) components, as described in Equations (2) and (3), respectively. In addition, an F_1 score of 87.5 is obtained according to F-measure, the metric considering both recall and precision, as described in Equation (4). In addition, the model was quantized with an unsigned integer of 8 bits for activation, integer of 8 bits for weight, integer of 32 bits for bias, and unsigned integer of 12/8 bits for the mult/shift scale, as shown in Table 2. Besides, the presented model has shown successful performance of qualitative evaluation, as shown in Figure 10, which demonstrates that the model can detect objects in diverse conditions.

$$Recall = \frac{TP}{TP + FP} \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

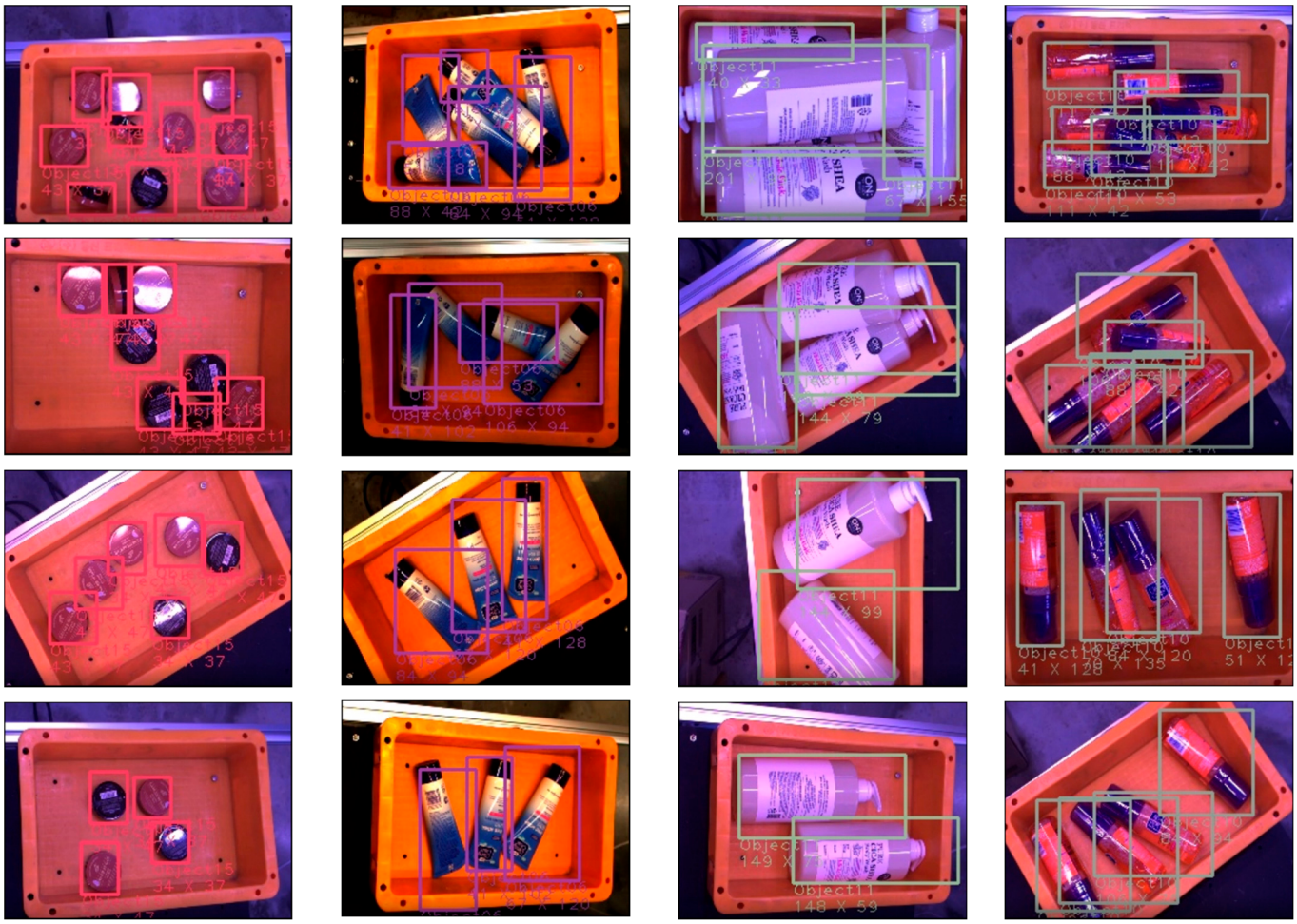
$$F_1 = 2 * \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

Table 1. Quantitative evaluation of the performance on object detection of the proposed model with an IoU threshold of 0.5 and a detection threshold of 0.697.

TP	FP	FN	GT	Recall	Precision
12,143	2561	898	13,041	93.1%	82.6%

Table 2. Quantization information for each component in the proposed model.

Activation	Weight	Bias	Scale
uint 8	int 8	int 32	uint 12/8 (mult/shift)



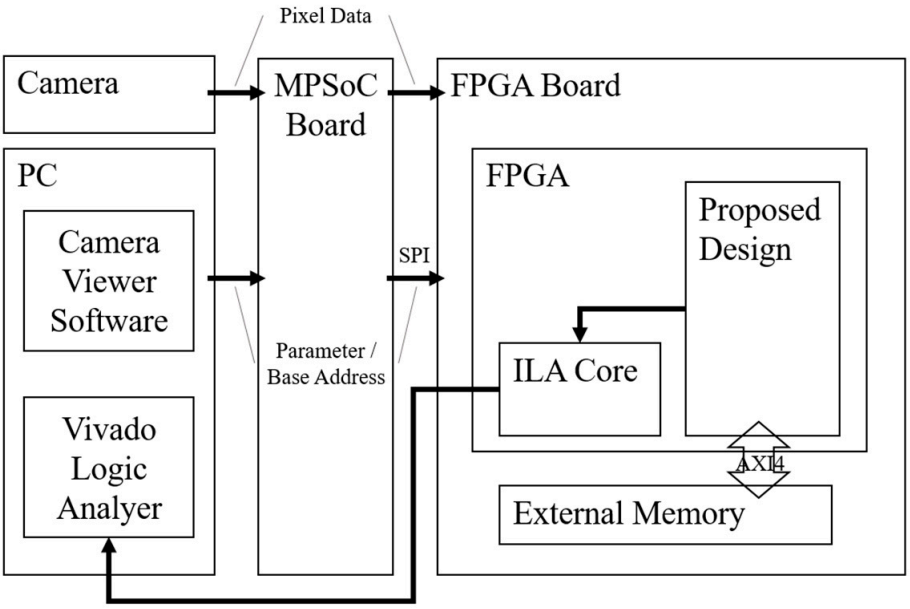
(**a**) (**b**) (**c**) (**d**)

Figure 10. Qualitative evaluation of the performance on object detection of the proposed model. There are different objects in horizontal, and different number of objects and angle in vertical: (**a**) small objects with red and black color; (**b**) middle size of objects with blue, white, and black color; (**c**) big size of objects with white color; (**d**) middle size of objects with pink and purple color.

5.2. Implementation Results and Comparison

The hardware based on the architecture proposed in Section 4 was designed using Verilog hardware description language (HDL) and verified by the coincidence of results between the C model of the presented model in Section 5.1 and register transfer level (RTL) simulation. Furthermore, it was implemented on an FPGA and verified in KU085 with the C model.

The system configuration for the experiment is illustrated in Figure 11. The camera sends pixel data to the MPSoC board, resizing the image to the size required by the model on the FPGA. In addition, the camera viewer software in the PC provides the parameter and base address to the MPSoC board and sends them to the FPGA using a serial peripheral interface (SPI). The proposed design on the FPGA starts to operate by fetching pixel data and parameters from an external memory on the FPGA board, and the design is verified by a Vivado logic analyzer utilizing an integrated logic analyzer (ILA) on the FPGA.



(**a**)

(**b**)

Figure 11. System configuration for the experiment: (**a**) block diagram; (**b**) hardware verification.

Table 3 presents a performance comparison with related works. The hardware resource utilization, particularly in BRAM and DSP, in [35] is quite high compared to those of [37], leading to high power consumption according to the total on-chip power in [38]. The BRAM and DSP consumed approximately 60% of the total power. In addition, the runtime is so slow that it is impossible to deploy a deep learning model on this hardware in real time owing to the performance of only 1 fps, although the logic size is also large in contrast to [37]. Consequently, the energy consumption is too high, making it unsuitable for edge devices.

Table 3. Performance comparison with related works.

		[35]	[36]	[37]	[38]	**Proposed** [1]
Platform		ZC702	VC709	ZC702	VC709	ZC702
Frequency (MHz)		-	110	100	100	100
Resource Utilization	LUT	54 k (102%)	324.7 k (75%)	20.2 k (37.8%)	83.6 k (19.32%)	18.3 k (34.5%)
	LUTRAM	-	-	1.2 k (7.3%)	-	5 (1%)
	FF	51 k (48%)	315.4 k (36%)	29.5 k (27.7%)	135.5 k (15.47%)	21.5 k (20.2%)
	BRAM	226 (80%)	2.7 k (92%)	134.5 (96.1%)	1.8 k (61.77%)	31.5 (23%)
	DSP	209 (95%)	1.8 k (53%)	192 (87.2%)	2.6 k (73.8%)	7 (3%)
Power (Watt)		7.95	27.7	2.23	8.9	2.11
Latency (ms)		1030	3.65	223.18	4.02	22.75
FPS (frame/sec)		1	273.97	2.62	248.76	43.95
Energy (mJ)		8188	101.11	497.6	35.78	48

[1] Results in ZC702 for a fair comparison with [35,37].

To deploy deep learning hardware accelerators on edge devices, the hardware size must be small with a lightweight design owing to limited resource constraints. However, the hardware size of [36] and [38] is too large over thousands for all components, 324.7 k and 315.4 k as LUT, FF for [36], and 135.5 k as FF for [38] in VC709. The BRAM and DSP utilization is also high, so their power consumption is quite high as 27.7 W for [36] and 8.9 W for [38]. They are not able to be embedded in resource-constrained edge devices because of an enormous amount of hardware utilization and cannot be operated on battery-operated IoT devices because they require high power consumption.

The hardware utilization for [37] in ZC702 is reasonable to be executed on low-cost devices, but the latency performance is quite poor. Therefore, it performs at only 2.62 fps, and it cannot be deployed in real time on IoT devices. Furthermore, its energy consumption is 497.6 mJ owing to its slow run time. Battery-operated devices cannot endure this hardware, which consumes high energy even though it is capable of being embedded on edge devices.

However, the proposed hardware design has extremely low utilization in BRAM and DSP owing to a memory-efficient on-chip memory management strategy and simple but powerful 3D tensor-like PE structure with a matrix generation scheme of the input feature map. Specifically, it achieves 1.25× and 4.27× smaller logic and BRAM size, respectively, and its energy consumption is approximately 10.37× lower than the previous low-cost hardware [37] with 43.95 fps as a real-time process under an operating frequency of 100 MHz on ZC702. This indicates that the proposed hardware can be embedded in resource-constrained edge devices and deployed in the battery-operated IoT devices for object detection in real time.

By applying parallel processing between layers and involving the maxpool function in convolution operation, latency is significantly reduced, as shown in Table 4. It was experimented in RTL simulations at conv6 and conv7 in an expand layer. This demonstrates that the proposed method achieves 2.19× faster latency than the conventional method of performing convolutions in series with an additional maxpool operation. In addition, the throughput performance was improved by 2.18× as well. It indicates that low-latency and high-throughput hardware with small size can be obtained with the proposed methods.

Table 4. Improvement of latency and throughput performance by applying parallel processing between layers and involving maxpool layer in convolution operation.

	Conventional	Proposed
Latency (us)	271.2	124.4
Throughput (activation/us)	90.6	197.5

Table 5 presents values indicating that the proposed hardware is capable of performing increased high-speed operations with an increase in its clock frequency, resulting in a higher fps performance as well. Based on the characteristics and environment of the devices to be utilized, the proposed hardware can be executed with optimal performance on such devices.

Table 5. Results of latency and fps performance according to various clock frequencies.

Frequency (MHz)	Latency (ms)	FPS (frame/sec)
167	13.62	73.4
200	11.3	88.5
220	10.15	98.5

6. Conclusions

With the development of deep learning technology and rapid growth in the area of IoT, deploying deep learning models on IoT devices has become an emerging field with TinyML. However, most deep learning algorithms are too complex to be executed on these resource-constrained edge devices. The algorithms are not suitable for battery-operated IoT devices owing to their high computation and energy consumption requirements. Therefore, a lightweight deep learning model and its well-optimized hardware through HW/SW co-optimization are required; hence, we proposed an optimized model for the hardware to be implemented and a lightweight and energy-efficient hardware architecture in this paper.

We presented an optimized model based on a backbone network by employing model simplification and compression. Model reduction techniques, such as involving maxpool operations in convolution, alleviates the computation complexity and latency, and hardware-oriented parameter simplification enables software and hardware to be co-optimized. Additionally, data quantization, as a model compression technique, was performed to reduce the storage space requirement for the parameters. Experiment results showed that the optimized model successfully performed object detection and was subjected to both qualitative and quantitative evaluations.

Furthermore, a lightweight and energy-efficient hardware architecture was proposed with a 3D tensor-like PE structure, generation of input feature map matrix, and a memory-efficient on-chip memory management strategy. The 3D tensor-like PE structure deals with several input feature map matrices at the same time, leading to low latency and eventually low energy consumption. In addition, logic size can be reduced owing to parallel processing between layers and channels through combination of the 3D tensor-like PE structure and input feature map matrix generation. Besides, the on-chip memory management strategy enables incremental access to an external memory without frequent and irregular data

read/write operations, leading to the use of a few and small on-chip memories, and also low power consumption owing to the simple access to the memories. As a result, the proposed hardware design achieves 1.25× and 4.27× smaller logic and BRAM sizes, respectively, and consumes approximately 10.37× less energy than those of previous similar works with 43.95 fps as a real-time process under an operating frequency of 100 MHz on a Xilinx ZC702 FPGA. It indicates that the proposed hardware is capable of being embedded in the resource-constrained edge devices and can be applied to the battery-operated IoT devices.

Author Contributions: Conceptualization, K.K.; methodology, K.K.; software, J.P.; validation, K.K., S.-J.J., J.P., E.L. and S.-S.L.; formal analysis, K.K., S.-J.J., J.P., E.L. and S.-S.L.; investigation, K.K., S.-J.J., J.P., E.L. and S.-S.L.; resources, S.-J.J. and S.-S.L.; data curation, K.K., S.-J.J., J.P., E.L. and S.-S.L.; writing—original draft preparation, K.K.; writing—review and editing, K.K., S.-J.J., J.P., E.L. and S.-S.L.; visualization, K.K., S.-J.J., J.P., E.L. and S.-S.L.; supervision, S.-J.J. and S.-S.L.; project administration, S.-J.J. and S.-S.L.; funding acquisition, S.-J.J. and S.-S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2022-0-00394, Development of AI semiconductor for level 4 self-driving based on functional safety).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Mijwil, M.M.; Abttan, R.A. Artificial Intelligence: A Survey on Evolution and Future Trends. *Asian J. Appl. Sci.* **2021**, *9*, 87–93. [CrossRef]
2. Schmidhuber, J. Deep Learning in Neural Networks: An Overview. *Neural Netw.* **2015**, *61*, 85–117. [CrossRef] [PubMed]
3. Deng, L. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Trans. Signal Inf. Process.* **2014**, *3*, e2. [CrossRef]
4. Bengio, Y.; Courville, A.; Vincent, P. Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* **2013**, *35*, 1798–1828. [CrossRef]
5. Bengio, Y. Deep learning of representations: Looking forward. In Proceedings of the International Conference on Statistical Language and Speech Processing, Tarragona, Spain, 29–31 July 2013; pp. 1–37.
6. Bengio, Y. Learning Deep Architectures for AI. *Found. Trends Mach. Learn.* **2009**, *2*, 1–127. [CrossRef]
7. Qijie, Z.; Tao, S.; Yongtao, W.; Zhi, T.; Ying, C.; Ling, C.; Haibin, L. M2Det: A Single-shot Object Detector Based on Multi-level Feature Pyramid Network. In Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019.
8. Zhao, Z.; Zheng, P.; Xu, S.; Wu, X. Object Detection with Deep Learning: A Review. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 3212–3232. [CrossRef]
9. Jeong, Y.; Son, S.; Lee, B.; Lee, S. The Braking-Pressure and Driving-Direction Determination System (BDDS) Using Road Roughness and Passenger Conditions of Surrounding Vehicles. *Sensors* **2022**, *22*, 4414. [CrossRef]
10. Andre, L.; Matthew, C.; Nathan, A.; Edwin, W.; Emil, D.; Bennie, V. Deep learning in the automotive industry: Applications and tools. In Proceedings of the IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 5–8 December 2016; pp. 3759–3768.
11. Chun-Hsian, H.; Po-Jung, C.; Yi-Jie, L.; Bo-Wei, C.; Jia-Xuan, Z. A robot-based intelligent management design for agricultural cyber-physical systems. *Comput. Electron. Agric.* **2021**, *181*, 105967.
12. Yanming, G.; Yu, L.; Ard, O.; Songyang, L.; Song, W.; Michael, S.L. Deep learning for visual understanding: A review. *Neuro-Comput.* **2016**, *187*, 27–48.
13. Liangzhi, L.; Kaoru, O.; Mianxiong, D. Deep Learning for Smart Industry: Efficient Manufacture Inspection System with Fog Computing. *IEEE Trans. Ind. Inform.* **2018**, *14*, 4665–4673.
14. Mehdi, M.; Ala, A.; Sameh, S.; Mohsen, G. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Commu-Nications Surv. Tutor.* **2018**, *20*, 2923–2960.
15. Xia, H.; Lingyang, C.; Jian, P.; Weiqing, L.; Jiang, B. Model complexity of deep learning: A survey. *Knowl. Inf. Syst.* **2021**, *63*, 2585–2619.

16. Hongfu, L.; Ziping, W.; Hengsheng, Z.; Bin, L.; Chenglin, Z. Tiny Machine Learning (Tiny-ML) for Efficient Channel Estimation and Signal Detection. *IEEE Trans. Veh. Technol.* **2022**, *71*, 6795–6800.
17. Lin, J.; Chen, W.M.; Lin, Y.; Gan, C.; Han, S. Mcunet: Tiny deep learning on IoT devices. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 11711–11722.
18. Disabato, S.; Manuel, R. Tiny machine learning for concept drift. *arXiv* **2021**, arXiv:2107.14759. [CrossRef]
19. Haoyu, R.; Darko, A.; Thomas, A.R. The synergy of complex event processing and tiny machine learning in industrial IoT. In Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, Virtual, 28 June–2 July 2021; pp. 126–135.
20. Simone, D.; Manuel, R. Incremental On-Device Tiny Machine Learning. In Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things, Virtual, 16 November 2020; pp. 7–13.
21. Landola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and < 0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
22. Alberto, M.; Rachmad, V.W.P.; Muhammad, A.H.; Muhammad, S. HW/SW co-design and co-optimizations for deep learning. In Proceedings of the INTESA '18: Proceedings of the Workshop on Intelligent Embedded Systems Architectures and Applications, Turin, Italy, 4 October 2018; pp. 13–18.
23. Park, S.; Kim, J.-J.; Kung, J. AutoRelax: HW-SW Co-Optimization for Efficient SpGEMM Operations With Automated Relaxation in Deep Learning. *IEEE Trans. Emerg. Top. Comput.* **2022**, *10*, 1428–1442. [CrossRef]
24. Li, Y.; Zhang, Y.; Zheng, W. HW/SW co-optimization for stencil computation: Beginning with a customizable core. *Tsinghua Sci. Technol.* **2016**, *21*, 570–580. [CrossRef]
25. Wang, C.-H.; Huang, K.-Y.; Yao, Y.; Chen, J.-C.; Shuai, H.-H.; Cheng, W.-H. Lightweight Deep Learning: An Overview. *IEEE Consum. Electron. Mag.* **2022**, 1–12. [CrossRef]
26. Lee, Y.J.; Moon, Y.H.; Park, J.Y.; Min, O.G. Recent R&D trends for lightweight deep learning. *Electron. Telecommun. Trends* **2019**, *34*, 40–50.
27. Duong, C.N.; Quach, K.G.; Jalata, I.; Le, N.; Luu, K. MobiFace: A Lightweight Deep Learning Face Recognition on Mobile Devices. In Proceedings of the 2019 IEEE 10th International Conference on Biometrics Theory, Applications and Systems (BTAS), Tampa, FL, USA, 23–26 September 2019; pp. 1–6. [CrossRef]
28. Agarwal, P.; Alam, M. A Lightweight Deep Learning Model for Human Activity Recognition on Edge Devices. *Procedia Comput. Sci.* **2020**, *167*, 2364–2373. [CrossRef]
29. Yatao, Y.; Runze, Y.; Longhui, P.; Junxian, M.; Yishuang, Z.; Tao, D.; Li, Z. A lightweight deep learning algorithm for inspection of laser welding defects on safety vent of power battery. *Comput. Ind.* **2020**, *123*, 103306.
30. Kim, S.-H.; Kim, J.-W.; Doan, V.-S.; Kim, D.-S. Lightweight Deep Learning Model for Automatic Modulation Classification in Cognitive Radio Networks. *IEEE Access* **2020**, *8*, 197532–197541. [CrossRef]
31. Jang, S.-J.; Kyung, C.-M. Resource-Efficient and High-Throughput VLSI Design of Global Optical Flow Method for Mobile Systems. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2020**, *28*, 1717–1725. [CrossRef]
32. Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H.-J. A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1861–1873. [CrossRef]
33. Pereira, P.; Silva, J.; Silva, A.; Fernandes, D.; Machado, R. Efficient Hardware Design and Implementation of the Voting Scheme-Based Convolution. *Sensors* **2022**, *22*, 2943. [CrossRef]
34. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
35. Arora, M.; Lanka, S. Accelerating SqueezeNet on FPGA. Available online: https://lankas.github.io/15-618Project (accessed on 7 July 2022).
36. Huang, C.; Ni, S.; Chen, G. A layer-based structured design of CNN on FPGA. In Proceedings of the 2017 IEEE 12th International Conference on ASIC (ASICON), Guiyang, China, 25–28 October 2017; pp. 1037–1040.
37. Mousouliotis, P.G.; Panayiotou, K.L.; Tsardoulias, E.G.; Petrou, L.P.; Symeonidis, A.L. Expanding a robot's life: Low power object recognition via FPGA-based DCNN deployment. In Proceedings of the 2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST), Thessaloniki, Greece, 7–9 May 2018; pp. 1–4.
38. Abd EI-Maksoud, A.J.; Mohamed, A.; Tarek, A.; Adel, A.; Eid, A.; Khaled, F.; Ibrahim, Z.; Mandouh, E.E.; Mostafa, H. FPGA Design of High-Speed Convolutional Neural Network Hardware Accelerator. In Proceedings of the 2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 23–25 October 2021; pp. 376–379.
39. Abotaleb, A.M.; Ahmed, M.H.; Fathi, M.A. SNAPE-FP: SqueezeNet CNN with Accelerated Pooling Layers Extension based on IEEE-754 Floating Point Implementation through SW/HW Partitioning On ZYNQ SoC. In Proceedings of the 2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 23–25 October 2021; pp. 400–404.
40. Sun, S.; Chen, W.; Wang, L.; Liu, X.; Liu, T.Y. On the depth of deep neural networks: A theoretical view. In Proceedings of the AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; Volume 30, pp. 5784–5789.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.