applied sciences

Article

Hardware/Software Co-Design for TinyML Voice-Recognition Application on Resource Frugal Edge Devices

Jisu Kwon [1] and Daejin Park [1,2,*]

[1] School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, Korea; kjisu96@knu.ac.kr
[2] School of Electronics Engineering, Kyungpook National University, Daegu 41566, Korea
* Correspondence: boltanut@knu.ac.kr; Tel.: +82-53-950-5548

Abstract: On-device artificial intelligence has attracted attention globally, and attempts to combine the internet of things and TinyML (machine learning) applications are increasing. Although most edge devices have limited resources, time and energy costs are important when running TinyML applications. In this paper, we propose a structure in which the part that preprocesses externally input data in the TinyML application is distributed to the hardware. These processes are performed using software in the microcontroller unit of an edge device. Furthermore, resistor–transistor logic, which perform not only windowing using the Hann function, but also acquire audio raw data, is added to the inter-integrated circuit sound module that collects audio data in the voice-recognition application. As a result of the experiment, the windowing function was excluded from the TinyML application of the embedded board. When the length of the hardware-implemented Hann window is 80 and the quantization degree is 2^{-5}, the exclusion causes a decrease in the execution time of the front-end function and energy consumption by 8.06% and 3.27%, respectively.

Keywords: TinyML; embedded system; field programmable gate array (FPGA); microcontroller unit (MCU); voice recognition; inter-IC sound (I^2S)

Citation: Kwon, J.; Park, D. Hardware/Software Co-Design for TinyML Voice-Recognition Application on Resource Frugal Edge Devices. *Appl. Sci.* **2021**, *11*, 11073. https://doi.org/10.3390/app112211073

Academic Editor: Kyungyong Chung

Received: 23 October 2021
Accepted: 19 November 2021
Published: 22 November 2021

1. Introduction

Machine-learning (ML) applications based on neural networks are expanding dramatically. A neural network consists of connections between layers and a weight indicating the connectivity between nodes. Machine-learning is used in various fields, and the size and depth of networks are increasing, which is necessary for the network to cope with complex inputs. Typically, when the number of layers increases in deep learning, the number of weights approaches billions. In large-scale ML, the computation must be processed on the server. However, edge devices that make contact with humans in the internet of things (IoT) are designed on the basis of microcontroller units (MCUs). MCUs used in typical edge devices have power consumption about uW/MHz, and hundreds of kilobytes flash memory. When software (i.e., firmware) is flashed into memory, MCU repeats the same operation until reprogramming. As the name of Edge indicates, edge devices focus on the role of sensing and collecting data through interactions with objects at the end of the IoT network. To collect vast amounts of real-world data, edge devices must be able to operate in various locations and under various conditions. Moreover, the available resources, e.g., memory and power, are meager, especially for IoT edge devices, because their accessibility is low and the environment is poor. As edge device moves away from the center of network to collect data generated from close interaction between real-world, there are constraints, such as insufficient power or a small memory size. Therefore, when running artificial intelligence applications on conventional edge devices, various methods are used to transmit the collected data to a resource-rich server to perform ML operations, and then retransmit the results back to the edge device.

In this paper, we focus on the preprocessing data overhead and the process of passing it as an input to the network model when an edge device executes an ML application. In a speech-recognition application, if raw audio sample data input from the microphone are transmitted directly to the model, then the waveform will contain too much data. Thus, excessive resources will be required for calculation. In addition, if different people pronounce the same word, they will have the same classification result. However, the waveform has different shapes when compared with the raw audio data waveform. Edge devices use preprocessing techniques, such as fast Fourier transform (FFT), to reduce the amount of computation and improve the model's classification performance. In this paper, we propose a technique that aims at improving the operation speed by allocating the preprocessing operations, previously performed using software, to custom-designed hardware, as shown in Figure 1. In order to prevent correct results from being obtainable because the syllables of a word are truncated by windowing in speech recognition, the accuracy increases as many iterative classifications are performed over a certain time interval. In the already existing application, the entire process of audio data collection, preprocessing, and classification by a model is performed in the MCU's software code. In this paper, an inter-integrated circuit-sound (I^2S) hardware module was custom designed for collecting and delivering sound data to an edge device. Moreover, the I^2S module performs part of the preprocessing process in the voice-recognition application. Preprocessing of raw audio data mainly uses digital signal processing (DSP) operations. With DSP operations, it is faster and more efficient to perform in parallel in hardware, which reduces the time taken for preprocessing audio data, thus allowing more classifications to be performed at the same time. In addition, energy consumption can be minimized if a custom I^2S module that is optimized for voice recognition is used. This approach is a software and hardware co-design for running ML applications efficiently in embedded system edge devices.

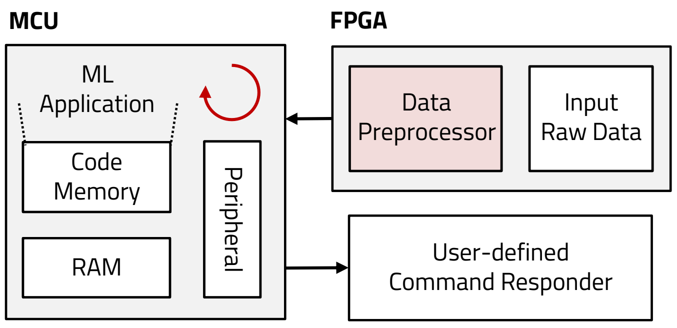


Figure 1. Proposed architecture overview for a custom hardware preprocessor coupled with a machine learning (ML) application.

To evaluate the proposed method, we used the edge device: an MCU board with a 32-bit ARM Cortex-M4 processor. The custom-designed DSP-embedded I^2S module was implemented in a Xilinx field-programmable gate array (FPGA). A TinyML program was modified as bare metal to implement an ML application on the MCU evaluation board. The audio data delivered to the edge device were transferred from the external FPGA via an I^2S communication protocol. The baseline for comparison is that the I^2S module delivers audio data, and the ML application, including the preprocessing of raw data, is run in the software. However, if preprocessing is included in the custom I^2S module, then the protocol used for data transfer is the same as I^2S, except that the data to be transferred have completed preprocessing. Moreover, the computation time and energy consumed in the two cases were compared.

The rest of the paper is organized as follows. Section 2 describes related works about the TinyML approach and audio recognition using ML. Section 3 presents the overall structure of the hardware and software co-designed audio-classification system. Section 4 describes the system's implementation and applies the proposed method to a word-classification application using an embedded board. Section 5 concludes the paper.

2. Related Works

Recently, an on-device ML paradigm is gaining attention in which the edge device itself processes data [1–4]. This in turn reduces the potential security problems when data are transmitted from the edge device to the server as well as the energy consumption generated from data communication.

Much research has already been conducted on the addition of dedicated hardware, such as a co-processor, to the central processing unit of a device where ML applications are executed. Furthermore, recent ML studies have focused on resource-rich servers or personal computers that support graphics processing units [5]. However, the conditions of the models used in a server differ from those used in an MCU. We note that, even if the model is only several megabytes in size, it will be difficult to use in MCUs [6]. However, various conditions must be considered before ML operations can be performed on resource-constrained edge devices, such as the size of the code to be written to the flash memory and the neural network layer's configuration. Therefore, the TinyML paradigm arose, and research is being conducted on performing ML applications in the MCU efficiently. TinyML is a paradigm that focuses on compressing neural network models, rather than complex inputs or performance, to enable ML applications on MCU-based edge devices [7–12]. Furthermore, TinyML allows the conversion of a model trained in an ML framework and programmed in a high-level language like Python into a C/C++ program by separating it into an interpreter and weights. This process involves quantization or precision reduction to fit the edge device's code-memory size and memory usage [13].

In addition, when ML applications are used in mobile devices that have lower resources and performance than PCs or servers, research is being conducted to accelerate ML applications using FPGAs or GPUs [14,15]. A mobile device with hundreds of megabytes of storage is sufficient to use the TensorFlow Lite framework's network output. Research on optimizing ML applications with the help of hardware such as FPGA or GPU is used more efficiently in mobile devices application processors which includes GPU inside [16,17]. A research approached from a framework perspective to accelerate neural network inference in mobile on-device, added a interface layer that can be accessed from mobile applications to TensorFlow Lite, allowing the GPU embedded in mobile devices to be used in inference of ML applications [18]. Furthermore, many studies have been conducted to accelerate the neural network model using FPGA, by implementing the entire neural network model on FPGA [19,20]. This approach resulted in reducing the time required for inference. The multiply-accumulate operation of a neural network may yield better results if it is performed in parallel in hardware than instruction by instruction executed in the processor. Other studies of TensorFlow Lite focused on performing ML applications by compressing the size of a deep learning model, which is optimized for PC resources, to use in mobile devices or minimizing amount of data communication with the cloud or server. However, neural network models for mobile devices cannot be directly applied to edge devices. Edge devices often lack the size of flash memory to store code, moreover absence of GPU. Therefore, in TensorFlow Lite for mobile, the TinyML approach for deployment to tiny edge devices should be used.

In our previous work, we proposed a prototype of a partial firmware replacement for a run-time network model [21–25]. This technique involves selecting a model corresponding to the input data domain from a memory-limited device. Another technique was described for automatically generating a suitable convolution neural network model [26–29]. This technique helps to overcome memory constraints in MCUs. Additionally, an efficient network suitable for edge devices is proposed from the same perspective, in which memory used for inference is saved using operator reordering. This approach focuses on reducing the memory used by the network model when the MCU runs ML applications. A technique for reducing the precision of parameters while minimizing the loss of network accuracy [30]. This technique helps reduce the memory usage occupied by ML applications.

The methods mentioned so far involve optimizing the embedded memory to match the MCU's conditions. In this paper, we focus on reducing the computation time and energy using a model with a memory size optimized for MCU and combining hardware. Additionally, several researchers combined hardware accelerators for ML applications [31]. However, only a few studies have been conducted at the level of embedded systems other than accelerators used with high-performance resources. In this paper, preprocessing hardware is used in the data-acquisition module during the data-input process, instead of accelerating ML computation.

3. Proposed Architecture

3.1. TinyML

TinyML can be defined as a combination of hardware and software algorithms for ML operating at the mW energy consumption range. TinyML requires an embedded system powered by a battery. Most frameworks for designing ML applications are implemented in Python, but this high-level abstraction language cannot be used in embedded systems. The independent blockchain used in embedded systems often supports only C/C++. Compiled code, called firmware, is written to the nonvolatile flash memory of the embedded system. This memory performs fixed operations until the firmware is updated. The TinyML framework is implemented in the C/C++ language for using ML applications in embedded systems.

The process of developing a TinyML application can be divided into two: generating a trained model and developing firmware to be executed in an embedded system based on the generated model. Neural network model generating and weight training are conducted at host machine, as shown in Figure 2b. In this paper, The TinyML framework used is TensorFlow Lite for MCU, which was released by Google [32]. When using TensorFlow in Python, typically on a PC or server, one should write code to configure and train the network architecture. Given that the output of TensorFlow is vast, it is impossible to apply that output to the limited memory of embedded systems. However, TensorFlow Lite is a framework for creating models in the range of hundreds of megabytes. These models can be used on mobile devices. Embedded systems can use small-sized models in the form of C/C++ arrays taking up several kilobytes created with the conversion functions provided within the framework. Additionally, TensorFlow Lite quantizes the weights to reduce the model's size, such as converting 32-bit floating-point weights to 8-bit integers. The model is used in the MCU-based embedded system after training in the TensorFlow Lite environment and then converted into a C/C++ array type. In the model's training process configuring the layer structure and evaluating the accuracy of the model's training is similar to the flow used in TensorFlow.

The next step is to develop the firmware that applies the model created as an array to the TinyML application. In the TinyML application, the model and interpreter are separated, and the interpreter executes the model in the form of a C/C++ array. The application's execution is described at the bottom of Figure 2c. The input data to be used in ML application may be unprocessed, like an image input to the convolution layer, or may require preprocessing, like audio waveform data. First, the input data are passed to the interpreter, which configures the number and types of layers of the neural network. Then the model is combined to perform inference on the input data. Once the inference is complete, the user must interpret the model's output and respond. When classification is performed on a single image, the class from the model's output layer can be interpreted as a result. These techniques can be mapped to the IoT layer. In the physical layer, the windowing preprocessing process of raw audio data is accelerated; In the communication layer, data is received from the external using the I^2S communication protocol; In the application layer, TinyML audio classification firmware is executed. However, in the case of an application that recognizes continuous speech, the inference is performed several times in a short time period to comprehensively interpret the output results. Finally, according to the results, the cycle is completed with the device in which the application is executed performing a predetermined response.

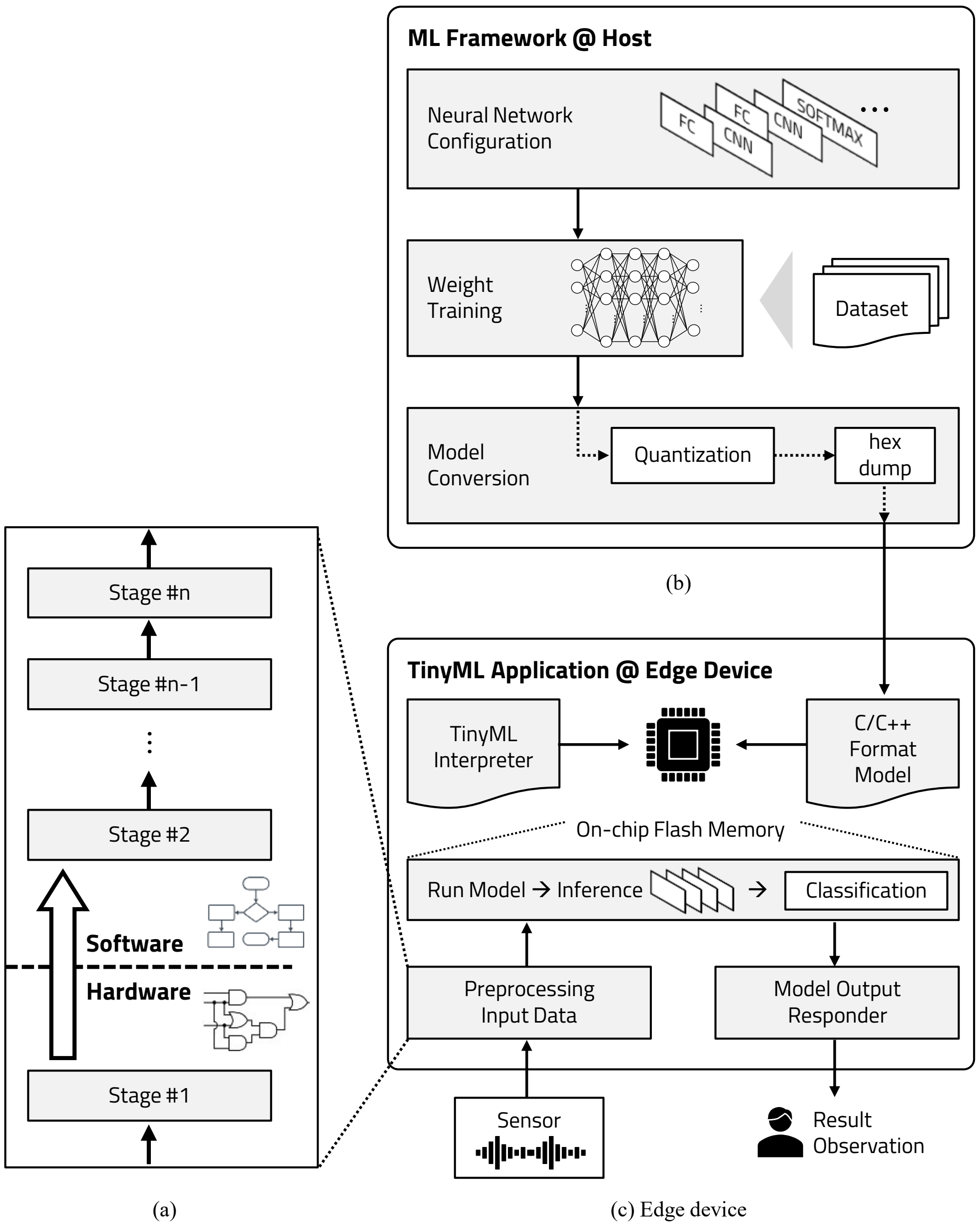


Figure 2. Overview of proposed TinyML application structure and design flow using machine learning (ML) framework at host. (**a**) hardware and Software partitioning at divided preprocessing stage (**b**) host machine (**c**) edge device.

In this paper, we focused on the aforementioned preprocess stage of the TinyML application in the edge device. When external data is input from the sensor, the software stored in the on-chip flash memory does not perform entire preprocessing, but partitions the part to be executed in hardware and the part to be executed in software. Considering the hardware characteristics useful in parallel data processing, some computations of preprocessing are conducted in hardware to reduce overhead.

3.2. Audio Classification

In this paper, the implemented TinyML application classifies words in audio data. Unlike image classification, finding words in continuous raw audio data requires preprocessing. Audio classification is an application that requires a lot of preprocessing of raw data. A typical sampling frequency of kHz-level produces a large amount of audio raw data over time. However, the network model used for classification cannot receive an input of that size. Therefore, features are extracted by using complex preprocessing in several steps. Among various audio classifications, the case used in this paper is word classification that recognizes specific words in the speaker's speech.

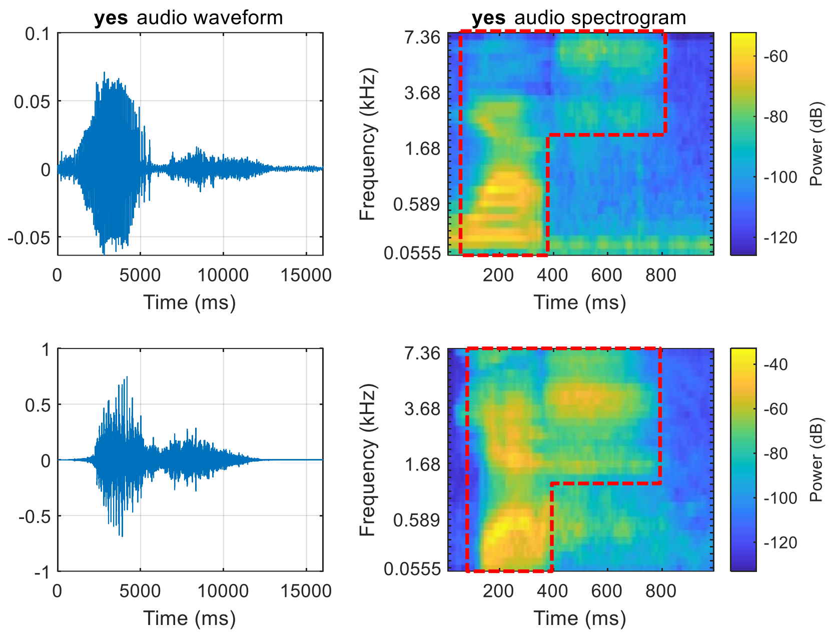
The word classification application specific words in a human's voice. Each person has a different voice frequency, amplitude, and pronunciation time. Therefore, it is necessary to find the target word's unique characteristics in the voice data sequence. Figure 3 shows the differences between the audio waveforms and spectrograms generated when different people pronounce the same word. Figure 3a shows the audio waveform when the word "yes" is pronounced, and the two-dimensional (2D) image is converted into a spectrogram. Figure 3b shows the audio waveform when the word "no" is pronounced. However, it is difficult to detect a word's unique characteristics from an audio waveform, but these characteristics can be visible on the spectrogram. The features of audio waveforms may differ even for the same word, according to the starting point of pronunciation. Additionally, the waveform's amplitude may vary depending on the person's voice. In contrast, regions with high power area in the spectrogram (yellow color) shows a unique shape according to each word in a 2D image of the same size. Unique shape of spectrograms are emphasized in red dotted line in Figure 3. For the word "yes", it shows a shape of ⌜ bent to the right, and the word "no", it shows a dense form on the left. In learning and inferencing in ML, a spectrogram can produce a better performance because these features are less affected by the characteristics of the human voice. Moreover, a spectrogram reduces the amount of computation performed in the neural network by compressing the raw audio data. Among the neural network structures, the convolution layer specializes in processing input images, so various applications use spectrograms to visualize audio. This audio is then used as input for the model.

The Figure 4 shows the process before the spectrogram is generated. When performing an FFT operation on a waveform, the signal at the window boundary becomes discontinuous. The discontinuities may include harmonic components that do not exist in the waveform during the FFT operation. Therefore, Hann windowing [33] is performed on the waveform, as shown in Equation (1). The maximum sample size is N.

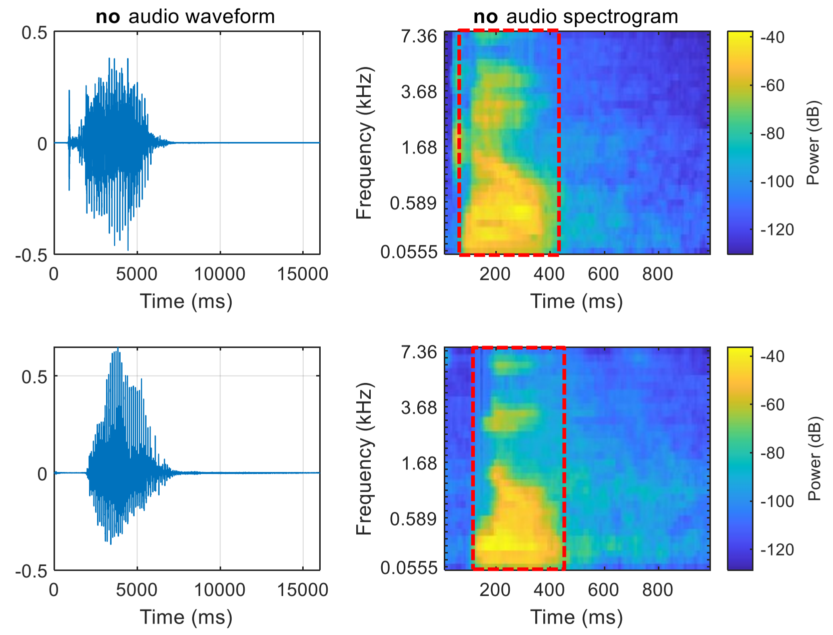
$$w(n) = 0.5\left(1 - \cos\left(2\pi\frac{n}{N}\right)\right), \quad 0 \leq n \leq N \tag{1}$$

Then, the mel-frequency scale, a nonlinear function, is applied to average the adjacent frequencies into the downsampled second array. This scale, as shown in Equation (2), gives more weight to low-frequency elements due to human sound characteristics, and it merges high frequencies. At this point, one row of the spectrogram is generated. Furthermore, the number of columns generated for the spectrogram equals the number of times this operation is repeated for an audio data sequence of a certain length.

$$mel\,scale = 2595 \times \log_{10}\left(1 + \frac{frequency}{700}\right) \tag{2}$$



(a)



(b)

Figure 3. Audio waveform and spectrogram comparison according to words. (**a**) "yes" word waveform and spectrogram (**b**) "no" word waveform and spectrogram.

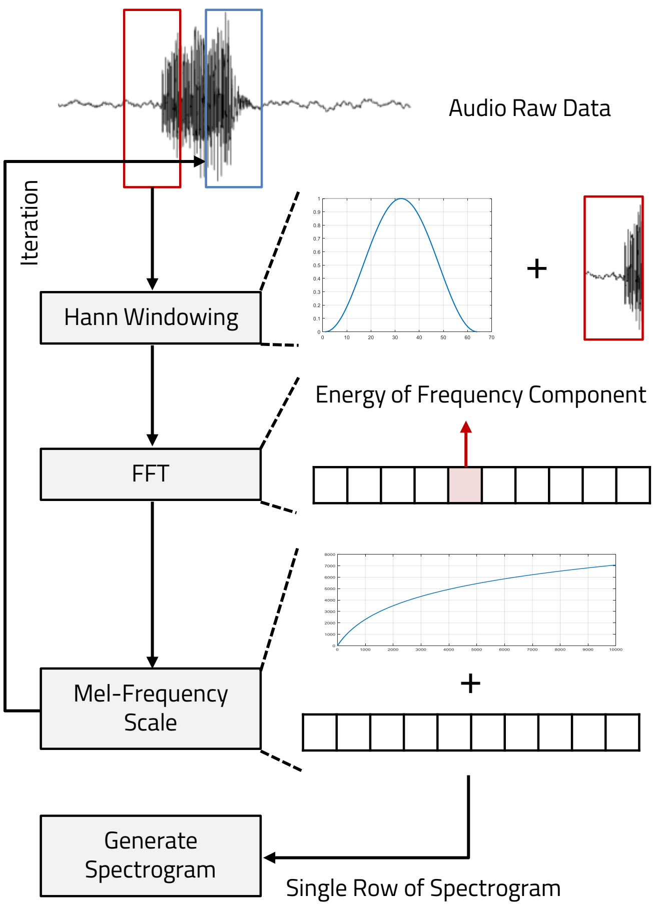
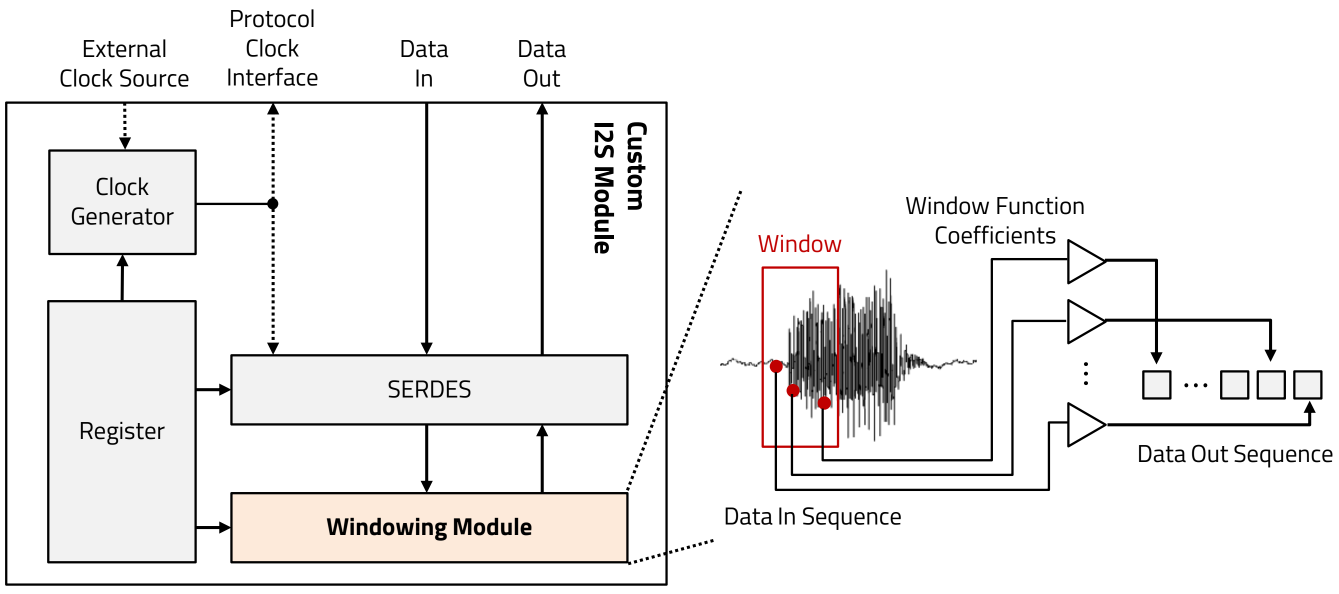


Figure 4. Overview of the audio data preprocessing to generate a spectrogram.

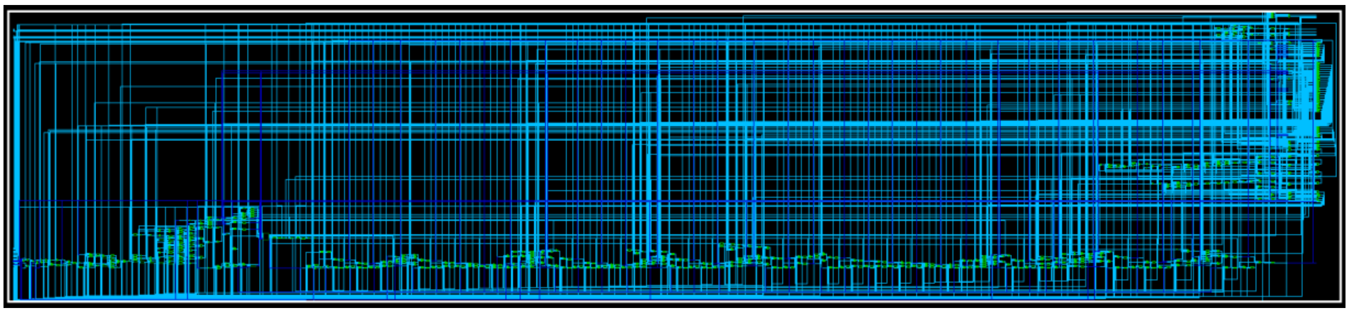
3.3. Custom I^{2}S Module

The edge device was assumed to be an embedded system with an MCU. It has operating frequency of about 100 to 200 MHz. Therefore, running a TinyML word recognition application in an edge device, it is necessary to extract features from continuous speech waveforms to reduce the amount of computation required to meet up low operation clock frequency. In the conventional approach, as a peripheral of the edge device, the hardware module for collecting voice data is considered separately from the ML software application. This approach makes a simple design, which is considered an advantage because the hardware and software must be designed and tested separately. However, if all of the computations and operations, except data collection, are processed via software, then the time required for such processes may increase due to the Von Neumann architecture. This architecture requires instruction fetching. Most of the audio-preprocessing operations for voice-recognition applications are DSP operations, such as FFT. Instead of complex instructions, it is more efficient to process DSP operations, which mainly are addition and multiplication operations, in hardware.

In this paper, the implemented I^{2}S module adds a windowing logic to the conventional structure using the Hann window coefficients, as shown in Figure 5a, which is a communication module for sound interfaces that support stereo or mono channels, and the module may be a master or slave. Figure 5b shows the synthesized Netlist schematic of the designed custom I^{2}S module. Additionally, the I^{2}S module is capable of both transmitting and receiving sound. This custom I^{2}S module is designed to perform a part of the windowing function in hardware during the word classification preprocess. Time and energy consumption overhead can be reduced by accelerating in parallel rather than in instruction fetch-based software. The proposed approach involves receiving a voice from an external microphone and transmitting it to the embedded system in which the TinyML application is executed. The I^{2}S module requires a serializer/deserializer module that deserializes the received data bit by bit and re-serializes the data for transmission. This requirement is because data transmission using the protocol consists of one line each for transmission and reception. The standard I^{2}S module does not manipulate any input pulse-code modulation audio data. However, our custom-designed I^{2}S module performs Hann windowing for each sample before transmitting the data received from the microphone. When the window boundary is determined, the sample matching the index of the Hann window function is multiplied by a coefficient, and the windowing is processed in resistor–transistor logic (RTL). The coefficient of the window function has a floating-point format, but in RTL, it is difficult to calculate the fraction. Hence, it is quantized as a sum of powers of 2. When data come in, they are multiplied by coefficients in order of sequence, and the results form an output sequence with a first-in, first-out policy. When running TinyML applications, if a part of the preprocessing process performed in the embedded system is distributed to hardware, then the logic area or power consumption may increase. However, if the increment is less than the resource savings in the embedded system, then it can be considered more efficient.



(a)



(b)

Figure 5. Proposed architecture of the custom I^{2}S module. (**a**) I^{2}S data path (**b**) synthesized schematics of I^{2}S implementation.

4. Experimental Results

The experiments with the proposed structure were conducted on an embedded board used to run the TinyML application and an FPGA, in which a custom I^{2}S module was programmed. The STM32F4-Discovery embedded board was used in the experiment as a TinyML application execution system based on ARM Cortex-M4 32-bit MCU. Table 1 shows the specifications of the board. The FPGA used an ARM Cortex-A9 processor and Xilinx 7-series combined Zynq-Z7 board [34]. The neural network used in the TinyML application on the embedded board was tiny conv [35,36].

The network consists of three layers. The first layer performs a convolution operation on a 49 × 40 size spectrogram image with eight 10 × 8 filters. In the second layer, the images generated by the filter are fully connected to the output layer. The third layer is the output layer, which has two trained words and silence and unknown classes. The softmax function was used to sharpen the differences between the results. The Speech Commands dataset was used to train the above network [37]. This dataset is open-source with over 100,000 WAV files of 1-s duration each. The word pair trained in this application was "*yes*" and "*no*".

Table 1. Specification of STM32F4-Discovery embedded board.

Core	ARM 32-bit Cortex-M4
MCU	STM32F407VGT6
Floating-point Unit	enable
Flash Memory	1 MByte
RAM	192 KBytes
Frequency	Up to 168 MHz

Afterward, FFT was applied to audio data to generate the first frequency array. The decision to recognize a word in the audio classification application used in this research was conducted at 1-s intervals. The overhead for audio preprocessing for the entire 1-s period is large. At a sampling frequency of several kHz, the size of data collected for 1-s is too large for signal processing such as FFT. Therefore, the small window is moved as much as the stride and frequency bucket. The result of Hann windowing and FFT operation, is stored as much as the operation result for 1-s.

If the period of decision is T when the window size is W and the stride size is S, then the number of frequency buckets is determined as the minimum i that satisfies Equation (3).

$$(W \times i) - (W - S) \times (i - 1) \leq T \tag{3}$$

The determined frequency bucket value affects the number of input layer nodes in the neural network.

Figure 6 shows the measurement of the ratio of functions corresponding to the front-end, specifically preprocessing, in the entire main loop when the TinyML application is executed on the embedded board. The front-end functions, which comprise 78.8% of the feature-generate function, include windowing, FFT, and log scaling, as described in Figure 4. The feature-generate function occupies 81% of the main loop. In this experiment, the windowing function will be handled in hardware.

The I^{2}S module was implemented as RTL on the FPGA board. Voice was input through a microphone connected to the I^{2}S module, and the embedded system received audio data using the I^{2}S protocol. Then, the audio data received by the embedded system was passed from the windowing function in the FPGA. A floating-point operation is required to calculate the coefficient values of the Hann window array. However, if a floating-point unit is added for windowing calculation, an excessive area will be required. Therefore, we quantized the coefficients as the sum of powers of negative 2. Using Equation (1), because of Hann coefficients have values from 0 to 1, the fractional part can be expressed as the sum of powers of negative 2. In this quantization technique, the error from the original coefficient varies according to the highest power of negative 2. As the number of negative power of 2 increases, the resolution of fractional parts that can be expressed increases, so the quantization error is small. The hardware usage is determined by the length of the Hann windowing function, i.e., the number of coefficients. To apply the Hann window sliding for each sample to the audio data input, coefficients need to be stored in the hardware. In the current implementation stage, the cell area after synthesis tends to be large. This increase in size is because the combinational and sequential logics are used without a separate storage device, such as RAM, for coefficients.

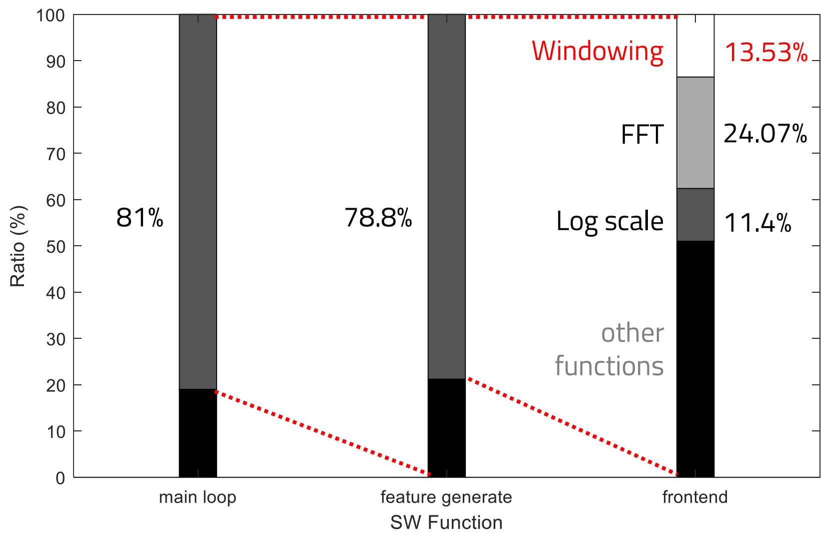


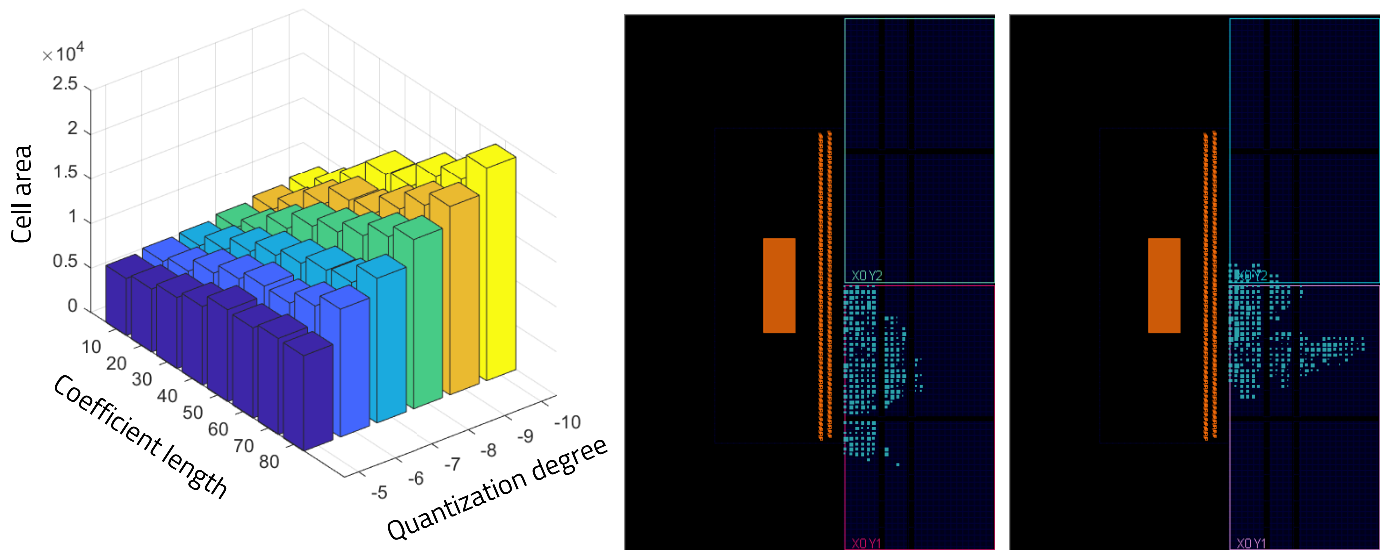
Figure 6. Function proportions of the TinyML application on the MCU edge device.

The Xilinx Vivado uses FPGA resources, which was placed on the chip beforehand, with routing when synthesizing and implementing RTL on Zynq FPGA. Utilization and area calculations may not be accurate due to different specifications and resources for each FPGA. Therefore, in this evaluation, the cell area was measured after RTL synthesis of the custom I^2S module using Synopsys Design Compiler.

Figure 7a shows the change in the cell area of the custom I^2S module while changing the number of coefficients, which was made according to the length of the Hann window and the maximum value of the number of the power of 2, according to the quantization precision. Figure 7b is the smallest FPGA tool synthesis result when the coefficient window length is 10 and the quantization degree is -5. Figure 7c is the largest case when the coefficient window length is 80 and the quantization degree is -10. Since the Hann window coefficient has a value from 0 to 1, the power of 2 is also negative. The number of coefficients was changed from 10 to 80 corresponding to the window size from 0.625 to 5 ms. Quantization degree represents the number of powers of 2 in the negative direction used to express decimal coefficients, and it was changed from -5 to -10.

As a result, the length and quantization degree of the coefficient (window) increased as the value of the synthesized cell area increased. To minimize the additional hardware usage, the coefficient length and quantization degree should be lower. However, if the window coefficient length factor is too low, the size of the window becomes smaller and the number of FFT operations performed in the MCU software, required to recognize a single word, increases. Alternatively, if the quantization degree is lowered and coarse-grained, it may affect the difference between the original Hann window function result. Considering this trade-off, it is important to determine a factor suitable for the application. In this paper, the Hann window array implemented in RTL in hardware has a length of 80, which corresponds to a window size of 5 ms at 16 kHz. This dimension matches those of the input layer of the neural network and the spectrogram-converted audio data sampling frequency. The quantization degree uses a value of -5; thus, the maximum resolution of the quantization is 2^{-5}.

Table 2 compares the changes in the execution time for the functions, the energy consumption of the embedded board, and the area after the synthesis of the RTL implemented in the FPGA board when the window process is excluded from the TinyML application using the proposed structure. In the row corresponding to the windowing function, a 0.219 ms decrease occurred, apart from the necessary time consumption for profiling. Furthermore, the execution time for other preprocessing functions also decreased slightly. Additionally, the energy consumption was reduced by 3.27%, as compared to the conventional case. Energy consumption was measured using the Microchip Power Debugger current visualizer. However, for the proposed structure, the hardware area of the I^{2}S module increased by 93.39%. The size of the window function for the audio data sequence affects the I^{2}S module's area increase. As the size of the window increases, the number of coefficients to be stored in the window function inevitably also increases. Therefore, it is important to select an appropriately sized window boundary according to the characteristics of the application to which the proposed structure will be applied.



(a) (b) (c)

Figure 7. Synthesis result cell area comparison according to Hann window coefficient length and quantize precision degree. (**a**) Design Compiler synthesis result according to coefficient length and quantization degreeu (**b**) FPGA synthesis result with the smallest cell area (**c**) FPGA synthesis result with the largest cell area.

Table 2. Comparison of time, energy, and the logic area between the conventional and proposed structures.

	Conventional		**Proposed**	
	ΔSystemClk	**Time (ms)**	**ΔSystemClk**	**Time (ms)**
Windowing	452,808	2.717	**416,412**	**2.498 (−8.06%)**
FFT	805,188	4.831	764,384	4.583 (−5.13%)
Log Scale	381,476	2.289	371,264	2.228 (−2.66%)
Frontend	3,345,885	20.075	3,251,249	19.507 (−2.83%)
Energy Consumption (mJ)	2.541		**2.458 (−3.27%)**	
Cell Area (gate)	55,350		**107,042 (93.39%)**	

5. Conclusions and Future Work

In this paper, we demonstrated efficient operation using both hardware and software to execute a TinyML audio classification in an embedded system using limited resources. The experimental results showed that the time and energy consumption decreased when executing the functions in the embedded board. In the audio classification application that goes through a lot of preprocessing on raw data input from external, we proved that co-designing with hardware and software reduces execution overhead rather than processing all operations with software. Previously, the I^{2}S module that received external audio raw data performed only the role of data collection. This paper designed a custom I^{2}S unit that added a windowing module, which is one of the audio preprocessing operations, and was able to reduce operation time and energy consumption in software. When designing the custom I^2S module, coefficient length and quantization degree were used as parameters for the windowing module. According to the combination, it was possible to design the custom I^2S module while minimizing the increase in the hardware area size. The proposed preprocessing technique using the hardware is suitable and can be applied in TinyML applications that require a lot of raw data preprocessing besides audio classification. In this paper, among the entire TinyML framework flow, software and hardware are partitioned based on the windowing operation. Optimized partitioning between hardware and software co-design may provide us with a new topic for future research.

Author Contributions: J.K. designed the entire core architecture and performed the numerical analysis; D.P. was the corresponding author. All authors have read and agreed to the published version of the manuscript.

Funding: This study was supported by the BK21 FOUR project funded by the Ministry of Education, Korea (4199990113966, 10%), and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF2019R1A2C2005099, 10%), and Ministry of Education (NRF2018R1A6A1A03025109, 10%), and Institute of Information & communication Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (no. 2021-0-00944, Metamorphic approach of unstructured validation/verification for analyzing binary code, 70%), and the EDA tool was supported by the IC Design Education Center (IDEC), Korea.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ML	machine learning
IoT	internet of things
MCU	microcontroller unit
AI	artificial intelligence
FFT	fast Fourier transform
I^2S	inter-IC sound
DSP	digital signal processing
FPGA	field programmable array
CPU	central processing unit
CNN	convolution neural network
PCM	pulse-code modulation
RTL	register transfer level
FIFO	first in, first out
SerDes	serializer/deserializer

References

1. De Prado, M.; Donze, R.; Capotondi, A.; Rusci, M.; Monnerat, S.; Pazos, N. Robust navigation with tinyML for autonomous mini-vehicles. *arXiv* **2020**, arXiv:2007.00302.
2. Kukreja, N.; Shilova, A.; Beaumont, O.; Huckelheim, J.; Ferrier, N.; Hovland, P.; Gorman, G. Training on the Edge: The why and the how. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 899–903.
3. Kolcun, R.; Popescu, D.A.; Safronov, V.; Yadav, P.; Mandalari, A.M.; Xie, Y.; Mortier, R.; Haddadi, H. The case for retraining of ML models for IoT device identification at the edge. *arXiv* **2020**, arXiv:2011.08605.
4. Disabato, S.; Roveri, M. 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, Online, 16 November 2020; pp. 7–13.
5. Luo, T.; Liu, S.; Li, L.; Wang, Y.; Zhang, S.; Chen, T.; Xu, Z.; Temam, O.; Chen, Y. DaDianNao: A Neural Network Supercomputer. *IEEE Trans. Comput.* **2017**, *66*, 73–88. [CrossRef]

6. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv* **2019**, arXiv:1801.04381.
7. Sanchez-Iborra, R.; Skarmeta, A.F. TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities. *IEEE Circuits Syst. Mag.* **2020**, *20*, 4–18. [CrossRef]
8. Ren, H.; Anicic, D.; Runkler, T.A. TinyOL: TinyML with Online-Learning on Microcontrollers. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China, 18–22 July 2021; pp. 1–8. [CrossRef]
9. Reddi, V.J.; Plancher, B.; Kennedy, S.; Moroney, L.; Warden, P.; Agarwal, A.; Banbury, C.; Banzi, M.; Bennett, M.; Brown, B.; et al. Widening Access to Applied Machine Learning with TinyML. *arXiv* **2021**, arXiv:2106.04008.
10. Banbury, C.; Zhou, C.; Fedorov, I.; Matas, R.; Thakker, U.; Gope, D.; Janapa Reddi, V.; Mattina, M.; Whatmough, P. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *arXiv* **2021**, arXiv:2010.11267.
11. Dutta, L.; Bharali, S. TinyML Meets IoT: A Comprehensive Survey. *Internet Things* **2021**, *16*, 100461. [CrossRef]
12. David, R.; Duke, J.; Jain, A.; Janapa Reddi, V.; Jeffries, N.; Li, J.; Kreeger, N.; Nappier, I.; Natraj, M.; Wang, T.; et al. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. *arXiv* **2020**, arXiv:2010.08678.
13. Soro, S. TinyML for Ubiquitous Edge AI. *arXiv* **2021**, arXiv:2102.01255.
14. Deng, Y. Deep learning on mobile devices: A review. In *Mobile Multimedia/Image Processing, Security, and Applications 2019*; International Society for Optics and Photonics: Bellingham, WA, USA, 2019; Volume 10993, p. 109930A.
15. Chen, Y.; Zheng, B.; Zhang, Z.; Wang, Q.; Shen, C.; Zhang, Q. Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions. *ACM Comput. Surv.* **2020**, *53*, 1–37. [CrossRef]
16. Zeroual, A.; Derdour, M.; Amroune, M.; Bentahar, A. Using a Fine-Tuning Method for a Deep Authentication in Mobile Cloud Computing Based on Tensorflow Lite Framework. In Proceedings of the 2019 International Conference on Networking and Advanced Systems (ICNAS), Annaba, Algeria, 26–27 June 2019; pp. 1–5. [CrossRef]
17. Ahmed, S.; Bons, M. *Edge Computed NILM: A Phone-Based Implementation Using MobileNet Compressed by Tensorflow Lite*; NILM'20; Association for Computing Machinery: New York, NY, USA, 2020; pp. 44–48. [CrossRef]
18. Lee, J.; Chirkov, N.; Ignasheva, E.; Pisarchyk, Y.; Shieh, M.; Riccardi, F.; Sarokin, R.; Kulik, A.; Grundmann, M. On-Device Neural Net Inference with Mobile GPUs. *arXiv* **2019**, arXiv:1907.01989.
19. Lacey, G.; Taylor, G.W.; Areibi, S. Deep Learning on FPGAs: Past, Present, and Future. *arXiv* **2016**, arXiv:1602.04283.
20. 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 2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST), Thessaloniki, Greece, 13–15 May 2019; pp. 1–4. [CrossRef]
21. Lee, D.; Moon, H.; Oh, S.; Park, D. mIoT: Metamorphic IoT Platform for On-Demand Hardware Replacement in Large-Scale IoT Applications (SCI). *Sensors* **2020**, *20*, 3337. [CrossRef] [PubMed]
22. Kwon, J.; Park, D. Toward Data-Adaptable TinyML using Model Partial Replacement for Resource Frugal Edge Device. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, Online, 20–21 January 2021; pp. 133–135.
23. Kwon, J.; Seok, M.G.; Park, D. Low-Power Fast Partial Firmware Update Technique of On-Chip Flash Memory for Reliable Embedded IoT Microcontroller (SCI). *IEICE Trans. Electron.* **2021**, *E104-C*, 226–236. [CrossRef]
24. Moon, H.; Park, D. Efficient On-Demand Hardware Replacement Platform toward Metamorphic Functional Processing in Edge-Centric IoT Applications (SCI). *Electronics* **2021**, *10*, 2088. [CrossRef]
25. Baek, J.; Jung, J.; Kim, M.; Kwon, J.; Park, D. Low-Power Metamorphic MCU using Partial Firmware Update Method for Irregular Target Systems Control (KCI). *J. Korea Inst. Inf. Commun. Eng.* **2021**, *25*, 301–307.
26. Fedorov, I.; Adams, R.P.; Mattina, M.; Whatmough, P.N. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. *arXiv* **2019**, arXiv:1905.12107.
27. Lawrence, T.; Zhang, L. IoTNet: An Efficient and Accurate Convolutional Neural Network for IoT Devices. *Sensors* **2019**, *19*, 5541. [CrossRef]
28. Liberis, E.; Lane, N.D. Neural networks on microcontrollers: Saving memory at inference via operator reordering. *arXiv* **2020**, arXiv:1910.05110.
29. Lin, J.; Chen, W.M.; Lin, Y.; Cohn, J.; Gan, C.; Han, S. MCUNet: Tiny Deep Learning on IoT Devices. *arXiv* **2020**, arXiv:2007.10319.
30. Rusci, M.; Capotondi, A.; Benini, L. Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers. *arXiv* **2019**, arXiv:1905.13082.
31. Mao, W.; Xiao, Z.; Xu, P.; Ren, H.; Liu, D.; Zhao, S.; An, F.; Yu, H. Energy-efficient machine learning accelerator for binary neural networks. In Proceedings of the 2020 on Great Lakes Symposium on VLSI, Online, 7–9 September 2020; pp. 77–82.
32. TensorFlow Lite. Available online: https://www.tensorflow.org/lite (accessed on 1 June 2020).
33. Essenwanger, O.M. *Elements of Statistical Analysis*; Elseview: Amsterdam, The Netherlands, 1986.
34. Digilent. Zybo-Z7. Available online: https://reference.digilentinc.com/programmable-logic/zybo-z7/start (accessed on 1 April 2021).
35. TensorFlow Source Code. Available online: https://github.com/tensorflow/tensorflow (accessed on 1 June 2020).

36. Warden, P.; Situnayake, D. *Tinyml: Machine Learning with Tensorflow Lite on Arduino and Ultra-Low-Power Microcontrollers*; O'Reilly Media: Sevastopol, CA, USA, 2019.
37. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv* **2018**, arXiv:1804.03209.