Received 10 August 2022; revised 5 November 2022 and 3 January 2023; accepted 8 February 2023. Date of publication 23 February 2023; date of current version 7 March 2023.

Digital Object Identifier 10.1109/OJCAS.2023.3245061

An On-Chip Fully Connected Neural Network Training Hardware Accelerator Based on Brain Float Point and Sparsity Awareness

TSUNG-HAN TSAI (Senior Member, IEEE), AND DING-BANG LIN

Department of Electrical Engineering, National Central University, Taoyuan 10617, Taiwan

This article was recommended by Guest Editor M. Rastogi.

CORRESPONDING AUTHOR: T.-H. TSAI (e-mail: han@ee.ncu.edu.tw)

This work was supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 111-2221-E-008-089-MY3.

ABSTRACT In recent years, deep neural networks (DNNs) have brought revolutionary progress in various fields with the advent of technology. It is widely used in image pre-processing, image enhancement technology, face recognition, voice recognition, and other applications, gradually replacing traditional algorithms. It shows that the rise of neural networks has led to the reform of artificial intelligence. Since neural network algorithms are computationally intensive, they require GPUs or accelerated hardware for real-time computation. However, the high cost and high power consumption of GPUs result in low energy efficiency. It recently led to much research on accelerated digital circuit hardware design for deep neural networks. In this paper, we propose an efficient and flexible neural network training processor for fully connected layers. Our proposed training processor features low power consumption, high throughput, and high energy efficiency. It uses the sparsity of neuron activations to reduce the number of memory accesses and memory space to achieve an efficient training accelerator. The proposed processor uses a novel reconfigurable computing architecture to maintain high performance when operating Forward Propagation and Backward Propagation. The processor is implemented in Xilinx Zynq UltraSacle+MPSoC ZCU104 FPGA, with an operating frequency of 200MHz and power consumption of 6.444W, and can achieve 102.43 GOPS.

INDEX TERMS Fully connected layers, on-chip training, optimized memory access, energy efficient, learning on edge, sparsity.

I. INTRODUCTION

DEEP neural networks (DNNs) have been incorporated in the field of computer vision [1], [2], and a fully connected layer (FC) has been employed extensively for image classification tasks. In the early stages, in pursuit of high accuracy for specific applications, researchers developed deeper network layers such as VGG16 [3] and ResNet [4], which have massive parameters. To achieve faster computation, GPUs have become indispensable for the inference and training phases of neural networks. Although GPUs are highly flexible, they also suffer from high power consumption and high latency. Due to the hardware architecture and compiler of the GPU, low performance and low utilization problems occur during the inference stage. Therefore, many DNNs accelerators for inference have been developed to solve this problem.

As shown in Fig. 1, the DNN model is trained on the server, and the trained network model is deployed on a dedicated processor for inference purposes. During the data transfer from the personal information server, there can be a data privacy problem, which may affect the user's privacy. In addition, when a large amount of data is transferred back and forth between servers and personal devices, it causes delays in transmission due to network speed factors, which affects the overall stability. Therefore, training on the edge

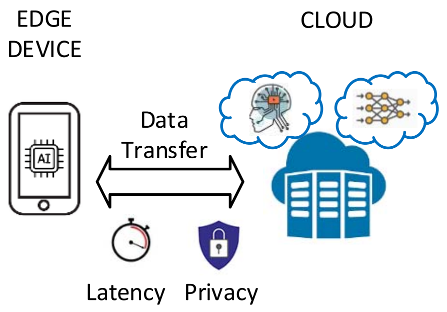


FIGURE 1. Training on cloud servers using personal data.

devices is essential to reduce the risk of data breaches and data transfer costs.

Several papers proposed DNN processors that can perform training to solve the privacy and latency problems encountered in inference-specific DNN accelerators [5], [6], [7], [8], [9], [10], [11], [12]. Due to the activation function module, many zero values are generated during the inference and training. These zero values are useless for computing operations, which occupy colossal memory space and affect the overall performance. Therefore, some papers proposed the use of the sparsity feature of DNNs for inference and training. For example, in an inference-only accelerator, Eyeriss [13] and EIE [14] use run-length coding (RLC) and compressed sparse column (CSC) compression methods to reduce the number of memory accesses and storage requirements of SRAM. Reference [7] introduced techniques for implementing sparse computation in the training phase to improve hardware performance. The edge device's computing and memory space are limited, so learning can only be done using small batches. According to [15], training with small batches (between 2 and 32) improves stability and convergence.

In DNN, Forward Propagation (FP) is the way to move from the input layer to the output layer in the neural network. It also used Backward Propagation (BP) for adjusting or correcting the weights to reach the minimized loss function. Due to the edge device's limited computing and memory bandwidth, multiple memory accesses are required to calculate the error/delta value of the backward propagation (BP) stage. This limitation leads to most of the on-chip training time being spent calculating error/delta values, creating a mismatch between the performance of FP and BP. Therefore, the processor used a memory-optimized access method proposed by Hussain and Tsai in [16] to speed up the error/delta calculation step. In the BP stage of the FC layers, the memory optimization method proposed in [16] can save 0.13x-13.93x memory accesses. It shows the advantage of FC layers for inference and training.

This paper proposes an energy-efficient sparsity-aware on-chip training processor. The proposed processor can perform FC layers-based inference and training on-chip. In this paper, we use Brain Floating Point as the data format of our architecture. Brain Floating Point format is developed by Google. It combines the advantages of IEEE 754 single-precision floating-point computing and half-precision floating-point computing. It retains the single-precision 8-bit exponent and combines the half-precision with fewer mantissa bits. In the experience, using the Brain Floating Point format can slightly reduce the accuracy, but it can significantly save chip area and power consumption. Previous works [5], [8], [10], [17], [18] use single-precision floating-point or 16-bit fix-point to train or inference neural networks. It will either spend lots of hardware resources or the accuracy loss on training and inference may not be negligible. To the best of our knowledge, we are the first to design a hardware architecture that uses the Brain Floating Point format for all operations. It can maintain a certain level of accuracy during training while using fewer hardware resources.

Another contribution of our paper is that we exploit the data sparsity on input feature maps. We use the sparsity map index (SMI) matrix to efficiently compress both input data and weight information. Compared to other compression methods, it has a significant improvement in compression rate. Although some papers such as SparTen [19] and SIGMA [20] have proposed similar methods, both papers extract non-zero information from on-chip memory while ours store the data that will be computed in on-chip memories, which makes memory storage more efficient. Overall, both methods make our hardware efficiency higher and power consumption lower than the other works.

The main contributions of this work are summarized as follows;

1) We use a 16-bit brain floating-point format to represent a wide dynamic range of numeric values by using a floating radix point.

2) We use a novel reconfigurable processing element (PE) architecture to complete the training and inference stages of the FC layer.

3) We use the sparsity of neurons and combine the optimized memory access method to reduce the memory space and the number of memory accesses required for FP and BP computations.

4) We implement the whole accelerator on ZCU104. It can achieve 102.4 GOPS with 256 Multiply–Accumulate (MAC) units working at 200 MHz. The design is scalable to expand according to the PEs to achieve higher throughput quickly.

The organization of the rest of the paper is as follows. Section II reviews the literature related to training processors. In Section III, the implementation of the proposed hardware architecture is discussed in detail. Section IV includes the results and discussion of our design. In addition, this section also contains the results of comparing with other literature. Finally, the conclusions are provided in Section V.

II. BACKGROUND

This section introduces the DNN hardware accelerators, the fully connected layer networks, and well-known floating-point computing formats. The basis is that since the inference and training of the FC layer are used, the weights need to be updated during the training stage to allow the model to converge and obtain the final desired effect. It means the hardware precision format is important, and the trade-off between accuracy and hardware resources is significant.

A. RELATED WORKS ON HARDWARE ACCELERATORS

Deep neural network techniques are composed of complex data access and complex computation to achieve more efficient processing of neural network computations. Many researchers have designed ASICs that can perform neural network inference, including Eyeriss [13], EIE [14], Nullhop [21], etc. Researchers also target FPGAs for neural network inference [22], [23], [24]. A hardware accelerator for inference is mainly optimized for the forward propagation of numerical computations in convolutional networks and fully connected layers, mainly for scheduling optimization of data in memory, reducing the number of memory reads as much as possible and improving the overall hardware performance by parallelizing the processing.

Among the research on hardware accelerators for ASIC reasoning, the MIT team is best known for their invention of Eyeriss [13] AI chip. It proposed a row stationary and used RLC, a compression format, in its designed architecture. Reference [25] proposed a hardware accelerator design for Angel-Eye. It constructs a programmable and flexible architecture for accelerators through data quantization strategies and compilation tools and illustrates the entire hardware design process. It first obtains parameters from trained models and quantizes them. It supports multiple neural network architectures through a compliable hardware architecture to increase flexibility, and finally maps them to the hardware for execution. Reference [26] proposed CNN accelerator designed on Xilinx Vertex 7 FPGA.

However, the above-mentioned hardware accelerators are designed for inference only and cannot perform the training of DNN models. To fine-tune the DNN model for such accelerators, the data is always transferred to the servers for the training, and the updated trained model is deployed on the edge device again. This leads to data privacy issues due to data transfer, which requires high bandwidth for high-speed data transfer.

Multiple hardware accelerators have been proposed to perform both training and inference of different DNN layers at the edge to deal with the data transfer issues during the training stage. Reference [5] proposed the F-CNN configurable training framework. It covers the training tasks of each layer of the CNN by reconfiguring the data flow path at runtime. Reference [6] uses a particular memory management unit to reduce the number of memory accesses during training. Experiments at different network sizes demonstrate the great flexibility of the proposed framework. Sticker [7] has been proposed to achieve high throughput for sparse FC layers. Liu et al. [8] designed an FPGA-based training accelerator using a unified computational engine and a scalable framework. An online learning processor has been proposed in [9] by Han et al. to support the training of the fully connected layers. It can perform object tracking and update the DNN model according to the changes in the data. EILE [10] was proposed to achieve incremental learning at the edge, where only the fully connected layer needs to be trained in its incremental learning processor. In [11], an FPGA-based accelerator with a compressed training scheme and effective compression using both quantization and pruning methods is proposed, exploiting sparsity in both forward and backward propagation. FPDeep [12] proposed a design framework for DNN training on multiple FPGAs in a cluster. In their research, the energy efficiency of training 16-bit AlexNet can be improved by a factor of 3.4 when the computations are distributed in a pipelined fashion across 15 FPGAs. Shao et al. [27] designed a reconfigurable processing element with a unified architecture that can flexibly support various computational modes during training and introduced scaling and rounding schemes to reduce memory usage.

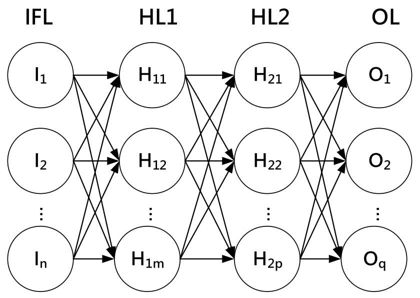


FIGURE 2. A small FC network made up of input features layer, hidden layer and output layer.

B. TRAINING ON FC LAYER

Fig. 2 shows a miniature model of the FC layer. It includes the input feature layer (IFL), hidden layer (HL), and output layer (OL). When the training process is performed in FC layers, there are three stages: FP, BP, and weight update (WU). FP starts from IFL, multiplies each node and weight, and accumulates the result to get the value of each node in the next layer. The BP is performed by partial linear differentiation based on the value of each node of the FP to obtain the gradient of the weights and the delta/error values. The WU stage uses the weight gradient obtained during BP to update the weights for training purposes. The following equations define these stages.

$$\text{FP: } O_n = \sum_i H_i \times W_{in} \tag{1}$$

In (1), O_n is the output node of the next layer, H_i is the node of the input layer, and W_{in} is the weight value associated with the output node that generates the next layer.

$$BP\text{: } \frac{\partial L}{\partial Win} = (O_n - Y_n) \times H_i \tag{2}$$

$$\Delta H_i = \sum_n (O_n - Y_n) \times W_{in} \tag{3}$$

In (2) and (3), the weight gradient and delta values are calculated where *On* is the output value at the output node *n*, *Yn* is the target value of output node *n*, also known as a ground truth value. *Hi* is the node's value where the weight *Win* is connected in the previous hidden layer.

$$WU\colon W_{new} = W_{old} - \eta \frac{\partial L}{\partial Wold} \tag{4}$$

In (4), W_{old} is the previous weight value. L represents the FC layer's loss, which is calculated at the end of the FP stage during the training process. η is the learning rate. Depending on the optimizer function used during training, it can be a fixed number or a variable number.

C. FLOATING POINT FORMAT

In many DNN hardware accelerators, the fixed-point format is mostly used as the computational format for the entire architecture. Many recent works have replaced fixed-point numbers with floating-point formats. This section describes several well-known floating-point formats and describes the advantages and disadvantages of these floating-point formats.

In the previous, the floating-point representation followed the IEEE 754 floating-point standard format [28], defined by the International Institute of Electrical and Electronics Engineers (IEEE). This standard describes two formats, single-precision floating-point, and double-precision floating-point numbers. They can be expressed by the following:

$$F = (-1)^S \times 1.f \times 2^{e-b} \tag{5}$$

where s is the sign bit, f is the mantissa bit, and e is the exponent bit. The single precision format bias is 127 and the double precision format bias is 1023.

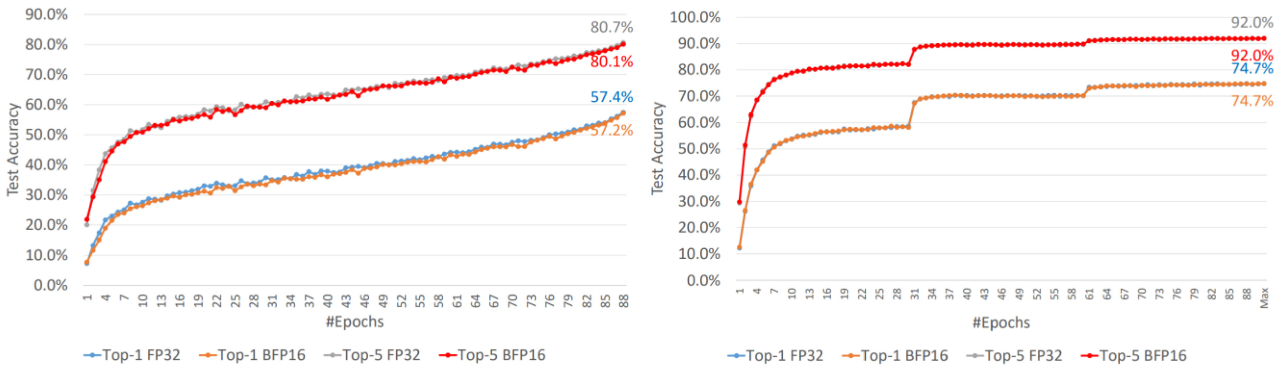
In recent years, many researchers have been working on accelerators for DNNs training, and as a result, many papers have proposed 16-bit or less than 16-bit floating-point computing formats. The IBM team has invented a new floating-point format and named it DLFloat [29]. DLFloat comprises 6 bits of exponential, 9 bits of fractional, and 1 bit of positive and negative numbers. Their definition is based on the actual range of values encountered in deep learning. Compared to IEEE 754 half-precision, the format has 1 bit more exponential and 1 bit fewer fractional bits. DLFloat also optimizes the floating-point format. Their proposed floating-point algorithm incorporates the NaN and infinity representations because when numerical operations are performed with NaN or infinity input, the result is always NaN or infinity. Therefore, they use $e = 63$ and $m = 511$ to represent the result of NaN and infinity, simplifying the logic unit of the FPU.

Flexpoint [30] is a floating-point computing format invented by the Intel team. Flexpoint combines the advantages of fixed-point and floating-point computing by using an index that automatically manages each tensor, using the same index for some operations to reduce computation and memory requirements. Flexpoint is tensor-based, using the N-bit mantissa to store the two's complement integer values and the M-bit exponent e shared among all tensor elements. This format is denoted as flexN+M. In general, the multiplication of two independent tensors can be computed as a fixed-point operation, which can turn most of the computations in deep neural networks into fixed-point operations. Flexpoint reduces memory and bandwidth requirements in hardware compared to single precision floating point. However, Flexpoint also has some disadvantages. Flexpoint is more complex than single-precision floating-point in format conversion and has a small dynamic range, which makes it easy to generate gradient disappearance problems when training neural networks, thus making it difficult for the model to converge.

TensorFlow-32 (TF32) [31] is a floating-point format proposed by NVIDIA to replace the single-precision floating-point format (FP32). TF32 uses the same 10-bit mantissa as the half-precision (FP16) math, shown to have more than sufficient margin for the precision requirements of AI workloads. And TF32 adopts the same 8-bit exponent as FP32 to support the same dynamic range. The advantage of TF32 is that the format is the same as FP32. When computing inner products with TF32, the input operands have their mantissa rounded from 23 to 10 bits. The rounded operands are multiplied exactly and accumulated in normal FP32. TF32 requires a CUDA compiler to perform the format conversion effectively. Although the dynamic range is the same as FP32, the complexity of TF32 will be more complex than other 16-bit floating point formats when designing the hardware, resulting in higher power consumption and area.

Finally, we introduce a 16-bit floating-point format developed by Google, called "Brain Float Point" [32], which consists of a 1-bit sign, an 8-bit exponent, and a 7-bit mantissa. This floating-point format combines the advantages of IEEE 754 single-precision floating-point computing and half-precision floating-point computing, which retains the single-precision 8-bit exponent and combines the half-precision with fewer mantissa bits. However, the accuracy is slightly less than that of single-precision floating-point computing. In hardware design, Brain Float Point uses fewer bits of mantissa bit, so it can significantly save chip area and power consumption. For example, using Brain Float Point will save eight times the power consumption for a multiplier than using single-precision floating-point computing. This is why Google and Intel use Brain Float Point as a floating-point format for their cloud servers. Brain Float Point has some advantages. It can directly intercept the first 16 bits of FP32, so it is straightforward to convert between FP32 and Brain Float Point. It also has a more extensive dynamic range than FP16, so it is less likely to overflow.

Although Google teams recommended that in general cases, representing activations in bfloat16 is generally safe, while weights and gradients should be kept in FP32 format. However, there is some potential to use bfloat 16 to represent more values. We trained AlexNet and ResNet-50 on ImageNet with all data formats where the computation



(a) AlexNet (b) ResNet-50

FIGURE 3. Validation accuracy for (a) AlexNet and (b) ResNet-50.

precision is set to bfloat 16. Fig. 3 shows that both top-1 and top-5 validation accuracy drops less or equal to 0.6%. Therefore, using bfloat 16 in all data formats is suitable for training and inferencing models and can significantly save hardware resources and power consumption.

Based on the above introduction, each floating-point computing format has its advantages and disadvantages. We prioritize the selection based on the ability to maintain a certain level of accuracy during training while using fewer hardware resources. In this paper, we use 16-bit Brain Float Point as the computational precision format for our hardware architecture.

III. PROPOSED WORK

In this section, we describe the proposed hardware architecture. In Section III-A, we present our overall architecture and data processing. In Section III-B, we present the data processing data flow. In Section III-C, the sparsity method in our design is introduced. The PE array architecture and the other Computational Core Unit (CCU) of the proposed design are explained in Section III-D. The Memory Bank (MB) of the proposed design is described in Section III-E.

A. OVERALL ARCHITECTURE AND PROCESSING FLOW

We use the SoC architecture which includes the PL (programmable logic) and PS (processing system) to design the hardware accelerator. The proposed accelerator is implemented on the PL side, and we use the AXI4 bus protocol to communicate with the PS. When PS needs to accelerate a fully connected neural network, it can control the accelerator on the PL side for inference and training of the neural network. PS can perform format conversion and data pre-processing of the feature maps and weight data. All feature maps and weights are stored in DDR4 on the PS side, waiting for PL to fetch these data. We use the DMA IP provided by Xilinx to implement high-performance burst transfers between PS DRAM and PL.

Fig. 4 shows the overall architecture of the proposed processor, and there are five main blocks in the architecture. These include a control unit (CU), memory bank (MB), computational core unit (CCU), data sparsity encoder and decoder unit, and a configuration register module to set the parameters used for training and inference stages, such as epoch, batch size, ground truth, etc. The data sparsity encoder and decoder unit are responsible for encoding and decoding the feature data. The MB stores the weights, input features, and output data generated by the CCU module. The CCU is responsible for processing the FP and BP calculations, including calculations such as output feature generation, activation function, softmax function, loss function, and weights update. The CU controls the data transfer from the processor, including data transfer from external memory to the MB for local storage, MB to the CCU for computation, and between different CCU modules during different calculation operations. ‘Module Gating’ is responsible for activating different modules to reduce the switching power of the processor. This is because during the processing of some modules, few of the modules remain inactive, e.g., PEs remain inactive when data is being transferred from off-chip memory to BRAM and BRAM to PEs, etc.

Fig. 5 shows the overview of the processor with two different kinds of data processing, i.e., inference mode and training mode. First, we read some information from the external memory into the configuration register for specification customization, and the related information to perform inference or training mode. In the case of training mode, the configuration register contains the information used in the inference stage and epoch, batch size, the number of images used for training, and the ground truth value of the images in the training mode.

In the inference mode, the non-zero input feature values are first stored in the input BRAM of the memory module by the data sparsity encoder module. The weight parameter is stored when all the input feature values are stored or the memory space is full. The weight values of the non-zero input features are stored in weight BRAM by the data sparsity decoder module. When the weights are stored to a certain amount, we transfer the input feature values and weight values to the computing core unit (CCU) to start the accumulation operation. In this stage, the data generated

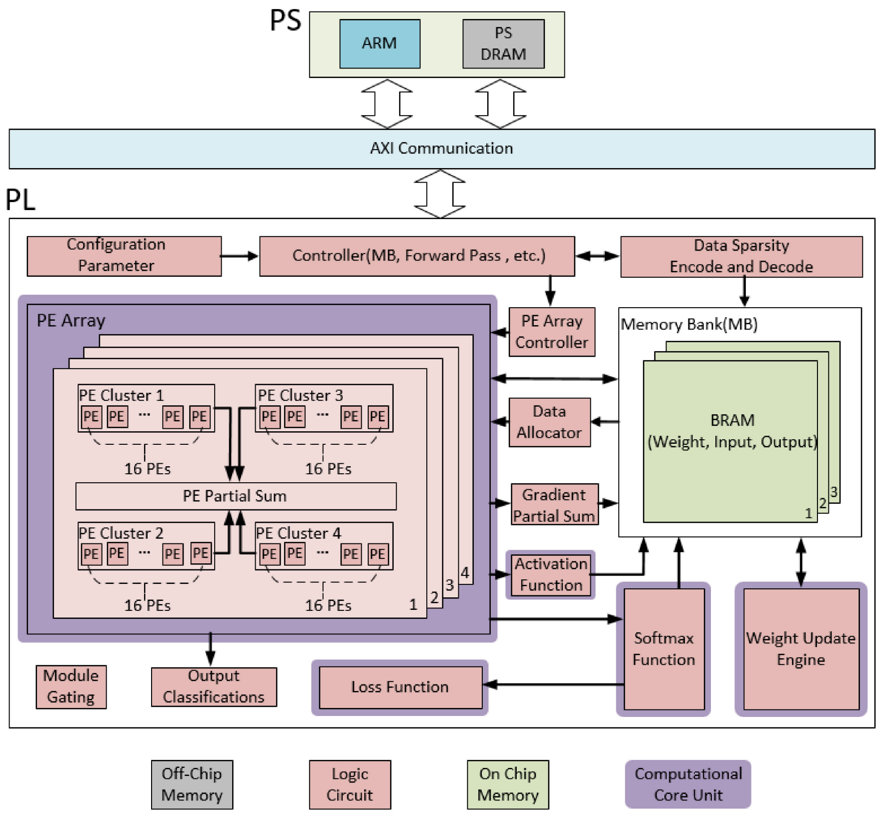


FIGURE 4. Proposed Overall Architecture.

by the computation will be transferred back and forth with the memory module to generate the node results of each layer. After the CCU completes the calculation of all FC layers, the data from the output layer is sent to the Output Classifications module for the final output classification, deciding which category it is finally classified.

In the training mode, the overall processing can be divided into the forward pass (FP) and backward pass (BP). The FP in the training mode is similar to the inference mode. The only difference is that the results in the output layer will be sent to the softmax module for calculation, and the BP stage will be started. In the next step, we will calculate the delta values and each weight gradient for each FC layer according to (2) and (3). The weight gradient for each layer is calculated cumulatively according to the size of the Batch Size. Therefore, the weight update will be calculated only when the specified Batch Size value is reached. When all the weights are updated, the newly obtained weights are used for the FP calculation again, and the above steps are repeated until the required number of updates is completed. The total number of weight updates (TWU) can be calculated as follows;

$$\mathrm{TWU} = \mathrm{E} \times \mathrm{ceil}\left(\frac{TI}{BS}\right) \tag{6}$$

where TWU is the abbreviation of Total Weight Updates, E is Epoch, TI is the total number of images used for training, and BS is the batch size.

B. FP AND BP PROCESSING DATA FLOW

Fig. 6(a) shows the data stream used by the accelerator for forward propagation. The proposed processor uses output stationary data flow to reduce the number of output data movements. If we repeatedly transfer the result of each computation to BRAM and then read it out from BRAM for accumulation, this step will cause a delay in data transfer and generate additional power consumption. Therefore, we store the results of each cycle in the partial sum registers in the PEs first, and then transfer the results to the memory module after all the values of that cycle have been computed. When processing the forward propagation data, the input feature data is distributed to each PE Cluster synchronously. As a result, each PE in the PE Cluster uses the same input feature data for computation while the weight data is distributed to each PE with different weights by Unicast.

Fig. 6(b) shows the data flow for the weight gradient calculation during the BP in the proposed processor. During BP, input stationary data flow reduces the number of input data moves. During BP, the input and weight data need to be reused for different computations. The input stationary method reduces data movement and provides data to PEs quickly to speed up BP. When calculating the weight gradient of BP, the result of each layer of FP and the error/delta values of each layer are used to generate the weight gradient value. This calculation also uses a mixture of unicast and broadcast modes to process the values.

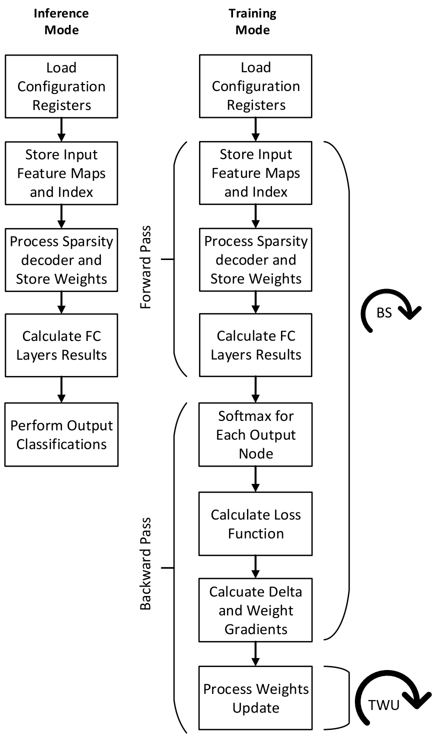

Inference Mode
Training Mode
Load Configuration Registers
Store Input Feature Maps and Index
Process Sparsity decoder and Store Weights
Calculate FC Layers Results
Perform Output Classifications
Softmax for Each Output Node
Calculate Loss Function
Calcuate Delta and Weight Gradients
Process Weights Update
Forward Pass
Backward Pass
BS
TWU

FIGURE 5. The data processing overview in the two different modes of the proposed processor. In inference mode, only multiplications and additions are performed to generate the final output results. In training mode, data processing complexity increases as the training mode includes processing of forward pass and multiple new computations, such as error/delta calculation, softmax, weight gradients, etc.

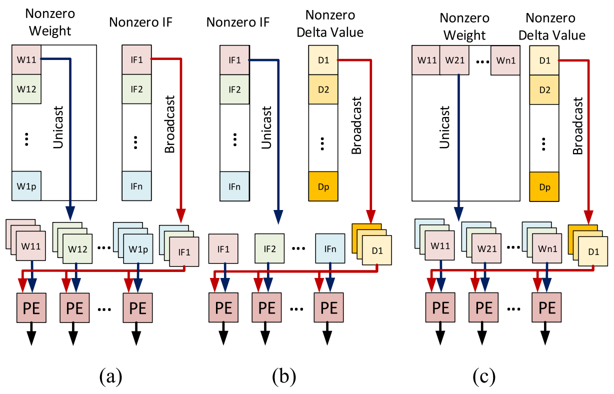


FIGURE 6. Proposed Data Processing Dataflow. (a) is Calculate Forward Pass, (b) is Calculate Weight Gradient and (c) is Calculate Delta value.

Fig. 6(c) shows the data flow used by the proposed processor to calculate the error/delta values. We adopt output stationery to reduce the time of output data transfers, power consumption, and memory access time. When calculating the error/delta values, the previous layer of delta values and weights is used, and unicast and broadcast mode generates the result values.

In the case of the forward pass, delta computation, and weight update, all three cases are more likely to reach memory bound because they all involved unicast mode to deliver data to PEs. Unicast mode will send different data to different PE which will potentially cause high memory

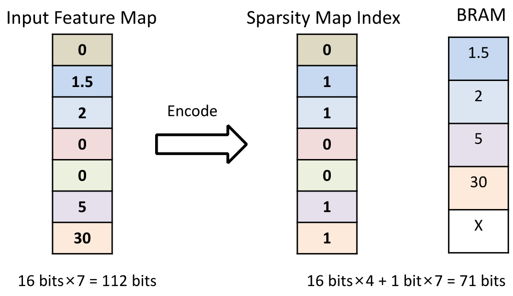


FIGURE 7. sparse matrix compression scheme.

bandwidth. Adding the sparsity feature can reduce the overall latency since it eliminates unnecessary operations, but the passes will still have potential memory bound when it comes to unicast mode.

When the number of MACs increases, if the number of memory is also increased, there will be no congestion problem since the memory bandwidth is also higher. If the number of memory remains unchanged, the performance of the accelerator will be limited because the value cannot be transferred to each PE in time when performing unicast mode.

C. DATA SPARSITY ENCODER AND DECODER UNIT

Our design uses a simple sparse matrix compression algorithm as shown in Fig. 7. The algorithm mainly modifies the CSC compression method. The original CSC algorithm requires three matrix spaces to store the values. The first matrix records the number of non-zero in each row, the second matrix records the location of non-zero values in each column, and the third matrix records non-zero values. Although CSC allows flexibility in processing data, it takes ample memory space to record the row and column coordinates of the values. Instead of using two matrixes to record the row and column coordinates position, we only use one matrix to save sparsity information. The sparsity map index (SMI) matrix can be generated using (7).

$$SMI(x, y) = \begin{cases} 1, & input(x, y) > 0; \\ 0, & input(x, y) = 0; \end{cases} \tag{7}$$

The values are coded as 0 or 1 according to the conditions in (7) and recorded in SMI. The non-zero input feature values are stored in the input BRAM of the MB. The weight values of the non-zero input features are stored in weight BRAM to save processing time in the inference and training stages through the sparsity table.

According to the above-proposed compression method, we use SMI, CSC, and Coordinate list (COO) to perform the relationship between sparsity and the input feature nodes, as shown in Fig. 8. When the sparsity of the input feature map is lower than 10%, the data transfer time will be longer than the original format and will have less reduction in PE computing time. The higher the sparsity is, the better the compression performance will be since both data transfer

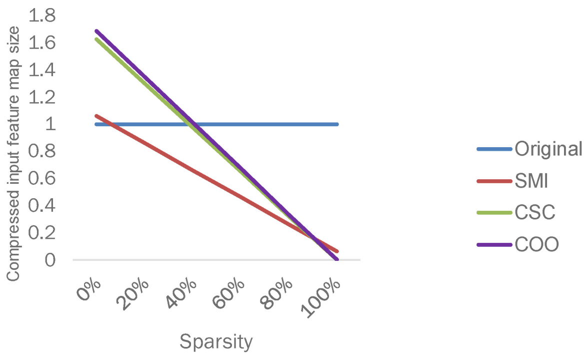


FIGURE 8. Comparison of compression methods over different sparsity amounts.

time and PE computing time are significantly reduced. Since the input features need to be flattened before the output nodes can be computed in the fully connected layer, the input feature nodes become one-dimensional matrices. CSC and COO need to record the positions of non-zero values. When the number of input feature nodes increases, more bits are required to record the positions of non-zero coordinates, which will cause additional capacity overhead. Therefore, using the proposed SMI compression format will have better compression results than CSC and COO.

According to the compression method proposed above, we can analyze the number of weight memory accesses for the forward propagation and backward propagation operations of the fully connected layer as calculated in (8) to (10):

$$\text{FP: } N_{HL2} \times \left[1 - sparsity(N_{HL2})\right] \times N_{OL} \tag{8}$$

$$\text{BP: } N_{deltaOL} \times \left[1 - sparsity(N_{deltaOL})\right] \times N_{HL2} \tag{9}$$

$$\text{WU: } N_{deltaOL} \times \left[1 - sparsity(N_{deltaOL})\right] \times N_{HL2} \times \left[1 - sparsity(N_{deltaHL2})\right] \tag{10}$$

Here, N_{HL2} is the total number of nodes in HL2, N_{OL} is the total number of nodes in OL, and sparsity () is the sparsity of the network layer.

From (8) to (10), we can conclude that using this bitmap compression method can reduce many memory accesses when performing FP and BP. Especially when the weight update is performed, the network sparsity is squared ratio with the number of memory accesses.

In our proposed architecture, we add the data sparsity encoder and decoder module before the memory bank to prevent the storage of zero values. These values are not worth further computation since the computation will always result in zero. These values include zero values in the input feature map and the corresponding weight values. The remaining values are stored in order. Hence, the frequency of accessing memory banks will be much lower, and the PE calculations will be reduced, which leads to lower computing and memory energy. Moreover, it will not affect the PE architecture since the data are already organized in the encoder and decoder modules.

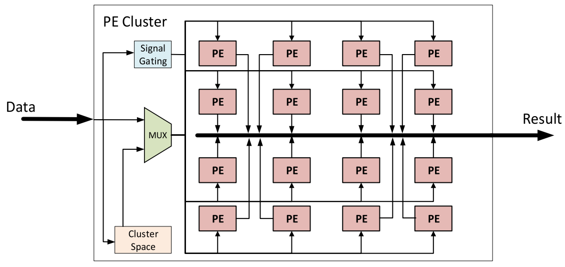


FIGURE 9. Architecture of PE Cluster.

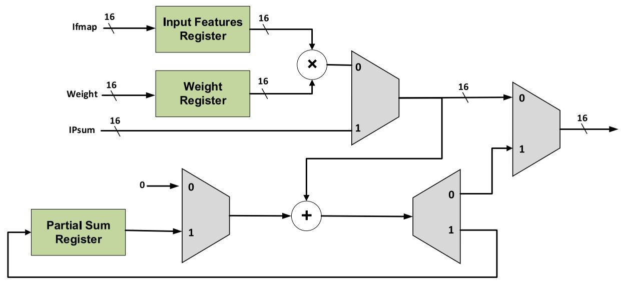


FIGURE 10. Architecture of the single PE.

D. COMPUTATIONAL CORE UNIT (CCU)

Fig. 9 shows the architecture of a PE Cluster where each PE Cluster is composed of 16 PEs. There are 16 PE clusters in the proposed processor. In each PE Cluster, all PEs process data in parallel; and the switching power is reduced by signal gating modules. Figure 10 shows the PE structure that can support both FP and BP. There is a total of 256 PEs in the proposed processor. As long as there are enough hardware resources, a PE array can contain more PE clusters and a PE cluster can contain more PEs.

The input and output of each PE are processed using the 16-bit brain floating-point precision format. Each PE comprises a multiplier, an adder, multiplexers, and a demultiplexer. A 4-stage pipeline is used to increase the operating frequency of the processor. Each pipeline stage consists of input data storage into registers, multiplication, addition, and output registers. Some registers are designed in the PE to reduce the memory accesses required for computing.

In image classification, the softmax function is used to map the value of the output layer between 0 and 1 to obtain the probability value of the category. The softmax function is defined as follows:

$$Softmax(x_i) = \frac{\exp(x_i - C)}{\sum_{j=0}^{K} \exp\left(x_j - C\right)} \tag{11}$$

Here, K is the total number of output classes in the output layer, x_i represents the output value of node I, and C is the maximum value of the output layer node.

From the above equation, it is known that the softmax is composed of complex mathematical operations. Compared to the previous hardware design using the RISC approach [33] and modified softmax function [34] to implement the softmax function, the proposed processor uses the fast exponential function proposed in [35] for exponential computation and a low resource divider module to complete

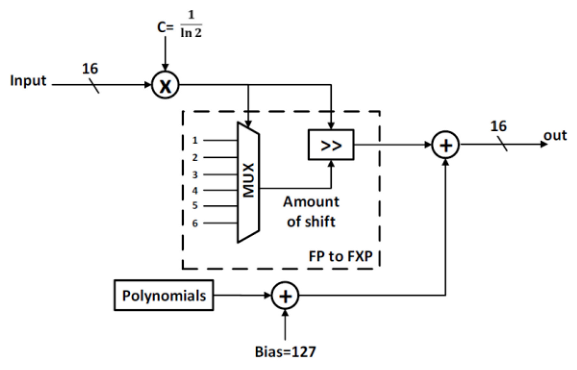


FIGURE 11. The hardware design of the exponent unit.

the softmax module. There are three modules in this softmax architecture, which are the exponential function module, the accumulator module, and the divider module. To enhance the operation frequency of the chip, the fast exponential function proposed in [34] is used to perform exponent calculation in the proposed design. The fast exponential algorithm is shown in (12).

$$e^x = (x \times \alpha + \beta - \gamma) \times (1 \ll 7) \tag{12}$$

Here $\alpha = 1.4426950409$, $\beta = 127$, and $\gamma = 0.0579848147$. The main purpose of α here is to convert the exponent to the second power of 2. β and γ are used to correct this exponent function. In the normal forward propagation process, output values produced by the output layer will be directly used to calculate the softmax function. However, the exponential function is too expensive for hardware implementation. Therefore, before performing the softmax function, the output layer will subtract its largest value to convert every value under zero. According to the characteristics of the exponential function, we can observe that when x is under zero, the function can be approximated as a linear equation. Since the softmax function will always redistribute all output values between 0 and 1, the approximation will not affect the prediction accuracy.

The hardware design for the exponent unit is shown in Fig. 11. It includes a single 16-bit multiplier, two adders, and one sifter in this architecture. The result of the exponent module is stored in the output BRAM after the accumulation (denominator of (11)). After all exponent calculations have been performed for all output nodes, the exponential values are read from the MB. The division unit performs the final division to get the final softmax result.

The division operation in (13) is performed after calculating the sum of the exponent values of the output nodes. Suppose O represents the exponent value of any given output node, and P represents the sum of all exponents of the output layer nodes. Then, the final softmax value (Q) for node E can be calculated as follows;

$$\mathrm{Q} = \frac{O}{P} \tag{13}$$

To reduce hardware resources, the division operation uses only shift and subtraction operations in pipelined mode.

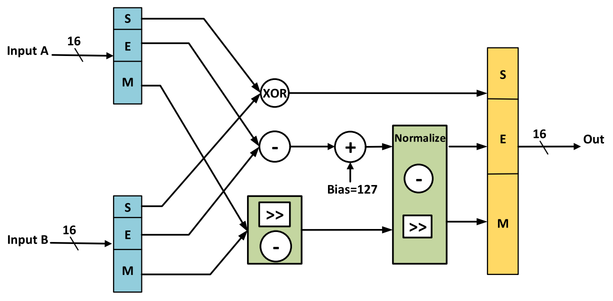


FIGURE 12. The hardware design of the division unit.

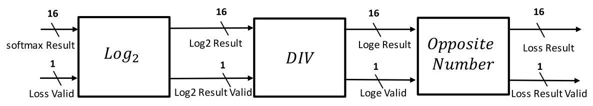


FIGURE 13. Block diagram for the hardware architecture of the loss function.

Fig. 12 shows the hardware architecture of the division module. In the sign bit of floating-point operation, we use the XOR logic gate to determine the positive and negative sign. In the exponent part, we mainly perform phase subtraction and then add bias. In the processing of mantissa, we use shift and subtractor to implement the divider. Our practice is similar to the concept of long division. Although the disadvantage of this is that it takes 20 cycles to get an output value, the advantage is to save hardware resources and achieves the effect of increasing the overall frequency.

Since the proposed processor is used for image classification tasks, the cross-entropy loss function used in the proposed processor can be defined as follows.

$$CE = -\sum_{i}^{C} t_i \ln(S_i) \tag{14}$$

where t_i is the ground truth value and S_i is the output value after the softmax function of each class in the C classes of the DNN model.

To obtain the loss values for the training stage, we calculate two values in this module: loss per round and loss per batch. Where loss per round is the loss value generated when training a DNNs model once. The loss per batch is the average loss value when training a specified number of times. For example, if batch size (BS) is equal to 4, then each round loss is generated four times, and per batch loss is generated using an average loss value based on each round loss. If the DNNs model is being trained for class C, then only one training image and one ground truth is provided per training round. It is known from (14) that for the class with zero ground truth, i.e., $ti = 0$, the loss value will be 0. Therefore, in the proposed processor, when $t_i = 1$, only one loss value is calculated for the single class in class C, which can save C-1 calculations of loss values in each training round.

Fig. 13 shows the overall block-level hardware architecture of the proposed cross-entropy loss processing. The input

Algorithm 1 Fast Binary Logarithm Algorithm

```
1. x ← Input Data
2. Initialize b=16'h3f00 and log_2_result=16'h0000
3. while(x<(16'h3f80))
     x←x+(1<<8)
     log_2_result ←(log_2_result -1)
4. z ← x
5. for (i=0; i < 7; i++)
     z ← $z^2$
     if(z>= (16'h4000))
       z ← z>>1
       log_2_result ← log_2_result +b
     b ← b-(1<<8)
6. Output log_2_result
```

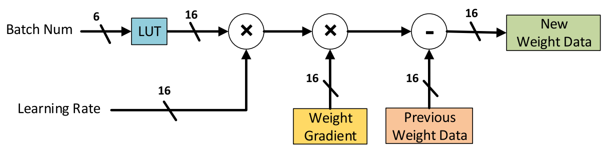


FIGURE 14. Proposed hardware design for Weight Update.

data from the softmax module results are in a 16-bit brain floating point format. In the first stage, the logarithmic function is computed in the "log2" module. In the second stage, the output of the log_2 unit is processed in the Divider module, which is a division of the log_2e parameter. In the final stage, we convert the result of the division output value by a positive and negative sign.

The fast binary logarithm [36] has been implemented for the high-speed design of *log* functions for loss calculation. The fast binary logarithm [36] benefits from high-speed and simple hardware architecture as it can be implemented efficiently with the pipelined-based design. Algorithm 1 shows the pseudo-code of a fast binary logarithm [36]. In Algorithm 1, precision represents the precision required for the decimal points. In the proposed design, 16-bit brain floating point numbers are supported where 1-bit sign bit, 8 bits are exponent, and 7 bits are mantissa bit. So the precision will be 7 based on mantissa for log calculation using Algorithm 1.

Figure 14 illustrates the weight update process in the WU module. The module is based on (4). First, the BS of the configuration register is converted to the reciprocal of the BS through the LUT register. For example, if BS = 4, the LUT conversion will result in a quarter (16'h3e80). Second, the result of the reciprocal is multiplied by the learning rate. Third, it multiples the accumulated weight gradient with the above result to get the updated weight error value. Finally, the old weight value is subtracted from the updated weight error value to obtain the new weight value. All computing units in this module use the sequential circuit to achieve high frequency.

E. MEMORY BANK (MB)

The memory bank (MB) is responsible for storing data transferred from external memory and different modules of the proposed processor. The MB is composed of 768KB of BRAM, and the memory capacity is divided into three main blocks to store different data types. The weight BRAM is 256 KB and is responsible for storing the weight values required for FP and error/delta calculations. The input feature BRAM is 256KB. It is responsible for storing the input features from external memory and the node results of FC layers and is also used to store the weight gradient when performing weight updates. The output BRAM is 256KB. It is responsible for storing the results of the CCU calculation, including the partial sum of each node of the FP, the weight gradient, and the error/delta value of the BP.

TABLE 1. Comparison of N-bit floating point add/subtracter.

N-Bits	Common Name	Power (mW)	Area (μm^2)
16 (1,5,10)	Half-Precision [28]	0.403	485.352
16 (1,6,9)	DLFloat [29]	0.348	474.919
16 (1,8,7)	Brain Float [32]	0.287	466.074
19 (1,8,10)	TF-32 [31]	0.352	554.299
22 (1,6,15)	Flexpoint [30]	0.737	772.027
32 (1,8,23)	Single-precision [28]	0.901	1134.907

TABLE 2. Comparison of N-bit floating point multiplier.

N-Bits	Common Name	Power (mW)	Area (μm^2)
16 (1,5,10)	Half-Precision [28]	0.508	641.844
16 (1,6,9)	DLFloat [29]	0.443	578.113
16 (1,8,7)	Brain Float [32]	0.315	447.476
19 (1,8,10)	TF-32 [31]	0.531	680.399
22 (1,6,15)	Flexpoint [30]	1.047	1185.257
32 (1,8,23)	Single-precision [28]	2.127	2157.321

IV. EXPERIMENTAL RESULTS

This section provides detailed experimental results and comparisons with other works.

A. COMPARISON OF FLOATING-POINT ARITHMETIC OPERATORS

Since we have investigated several floating-point computing formats, here we analyze these floating-point computing formats evaluated with their digital circuits. Our design environment is on the Design Compiler provided by Synopsys. Here we use TSMC 40nm process technology and 200MHz operating frequency for evaluation. Table 1 shows the synthesis results for floating-point adder/subtractor with different bit widths. The floating-point format for each bit width is represented by N(S, E, M), where N denotes the total number of bits, S is the sign bit, E is the exponent bit, and M is the mantissa bit. It shows that the 16-bit floating point format will get a smaller area and power consumption than single- and half-precision. Also, the brain floating point format can get the lowest area and power consumption.

Table 2 shows the synthesis results for floating-point multipliers with different bit widths. As shown in Table 2, the fewer the number of mantissa bits used in floating-point multiplication, the smaller the area and energy consumption. Also, the brain floating-point format has the smallest area and power consumption under the same condition of 16-bit length.

TABLE 3. DRAM access comparison between the original and use sparsity of AlexNet.

Layer	In/Out Channels	[16] Method (MB)	Proposed Method (MB)	Zero in Ifmap (%)	DRAM Access Reduction Ratio
FP-FC1	9216/4096	72.0254	26.654	63	63%
FP-FC2	4096/4096	32.016	9.61	70	70%
FP-FC3	4096/1000	7.8222	2.74	65	65%
BP-FC3	1000/4096	23.455	13.294		43.33%
BP-FC2	4096/4096	96.0234	17.933		81.37%
BP-FC1	4096/9216	144.0254	15.993		89.9%

TABLE 4. DRAM access comparison between the original and use sparsity of VGG16.

Layer	In/Out Channels	[16] Method (MB)	Proposed Method (MB)	Zero in Ifmap (%)	DRAM Access Reduction Ratio
FP-FC1	25088/4096	196.056	29.415	85	85%
FP-FC2	4096/4096	32.016	9.61	70	70%
FP-FC3	4096/1000	7.8222	2.35	70	70%
BP-FC3	1000/4096	23.455	12.512		46.66%
BP-FC2	4096/4096	96.0234	15.3725		84%
BP-FC1	4096/25088	392.056	17.649		95.5%

TABLE 5. DRAM access comparison between the original and use sparsity of MobileNet-V1.

Layer	In/Out Channels	[16] Method (MB)	Proposed Method (MB)	Zero in Ifmap (%)	DRAM Access Reduction Ratio
FP-FC1	1024/1000	1.957	0.784	60	60%
BP-FC1	1000/1024	3.91	1.565		60%

B. MEMORY ACCESS ANALYSIS

We adopt the technique proposed by [16] for reducing memory accesses in the backpropagation stage and combining it with a sparsity compression scheme. These techniques also reduce the number of external memory accesses. Table 3, Table 4, and Table 5 show the analysis of memory accesses for different layers in AlexNet [1], VGG16 [3], and MobileNet-V1 [37] respectively. DRAM accesses include the accesses to read input feature nodes, output feature nodes, weights, and weight gradients. We use the ImageNet [1] dataset to analyze DRAM accesses. Here, we set the batch size to 1 and epoch to 1. Table 3 to Table 5 shows that the fully connected layer has many parameters that need to be accessed from DRAM and sent to the PL side for numerical computation. Since the activation function by ReLU turns negative values to 0, we can reduce the number of DRAM accesses by skipping some weights with input values of 0. In forward propagation, the weights are only used to calculate the output of the hidden layer. While in backward propagation, the weights are used to calculate the delta value of each layer and get the new weight value so that more DRAM accesses are needed in backward propagation. Compared with [16], our proposed method can reduce DRAM accesses by 43.33% to 95.5% in forward and backward propagation. It can be seen that higher network sparsity leads to a higher compression rate, which results in better processor performance because fewer data needs to be accessed from DRAM. Since we reduce the DRAM accesses for some useless parameters, we can speed up the computation during the inference and training stage.

TABLE 6. Overall hardware specifications.

Parameter	Description
Working Frequency	200 MHz
Number of PEs	256
Power Consumption (W)	6.444*
Arithmetic Precision	16-bits BFP
Throughput (GOPS)	102.43
Energy Efficiency (GOPS/W)	15.895
Suppported Layers (Inference)	FC
Supported Layers (Training)	FC
Loss Function	Cross Entropy Loss
Activation Functions	ReLU
Maximum Epochs	Any number. No limitations
Batch Size	1~32
FC1	1~4096
FC2	1~4096
Maximum Outputs Calssification for normal DNN model	1000
Softmax Support	Yes

*: Include processing system (PS) power

TABLE 7. Resource usage of the proposed design on ZCU104.

Resource	Utilization	Available	Usage
LUTs	73622	230400	31.95%
LUTRAMS	983	101760	0.97 %
Flip-flops	26832	460800	5.82%
DSP	1285	1728	74.36%
BRAMs	220	312	70.51%
BUFGs	3	544	0.55%

C. OVERALL HARDWARE SYNTHESIS RESULTS

Table 6 shows the different attributes of the proposed chip design. We use 256 PEs and brain floating point precision format in the proposed design. Our design can support the inference and training mode for FC layers (FC1 and FC2). The FC1 and FC2 layers are designed in a reconfigurable way. They can support 1$\sim$4096 nodes, the output layer can support up to 1000 classes, batch size can support 1$\sim$32, and epoch can support any number. We also support softmax and cross-entropy loss functions so that we can end-to-end train the images for image classification.

The proposed design has been implemented in Verilog HDL and synthesized for Xilinx ZYNQ UltraSacle+MPSoC ZCU104 FPGA where we use Vivado 2020.1 as the development environment. The hardware utilization of the proposed design is shown in Table 7. Our design utilizes 73622 LUTs

TABLE 8. Layer by layer execution times for AlexNet on ZCU104.

Layer	Input size	Filter size	Output node	Parameters	Zero in Ifmap (%)	Time w/o sparse compression (ms)	Time w/ sparse compression (ms)	Reduction ratio	Average throughput (GOPS)
FP-FC1	256x6x6	1x1	4096	37,748,737	63	440.49	168.19	62.82%	88.24
FP-FC2	4096x1x1	1x1	4096	16,777,217	70	188.95	59.72	68.4%	87.8
FP-FC3	4096x1x1	1x1	1000	4,096,000	65	52.35	22.1	57.8%	88.44
BP-FC3	1000x1x1	1x1	4096	8,192,000		143.475	81.98	42.86%	92.53
BP-FC2	4096x1x1	1x1	4096	33,554,434		614.25	120.67	80.36%	92.91
BP-FC1	4096x1x1	1x1	9216	75,497,474		920.44	106.36	88.44%	88.78
Total						2359.955	559.02	76.32%	89.87

TABLE 9. Layer by layer execution times for MobileNet-V1 on ZCU104.

Layer	Input size	Filter size	Output node	Parameters	Zero in Ifmap (%)	Time w/o sparse compression (ms)	Time w/ sparse compression (ms)	Reduction ratio	Average throughput (GOPS)
FP-FC1	512x7x7	1x1	4096	102,760,448	85	1255.6	200.62	84.02%	88.81
FP-FC2	4096x1x1	1x1	4096	16,777,217	70	187.68	62.01	66.95%	87.72
FP-FC3	4096x1x1	1x1	1000	4,096,000	70	52.41	20.68	60.54%	89.34
BP-FC3	1000x1x1	1x1	4096	8,192,000		142.92	79.38	44.45%	92.49
BP-FC2	4096x1x1	1x1	4096	33,554,434		619.41	102.99	83.37%	92.99
BP-FC1	4096x1x1	1x1	25088	205,520,896		2495.32	114.12	95.43%	88.72
Total						4753.34	579.8	87.8%	89.36

and 26832 flip-flops. Moreover, the implementation design can achieve an operating frequency of 200MHz. We also use the Vivado Tool to analyze the power consumption of the proposed hardware architecture. The total power consumption was 6.444 W. The dynamic power consumption was 5.729 W, while the static power consumption was 0.715 W, and the power consumption of PS accounted for 45% of the dynamic power consumption, followed by 19% of the logical computing unit. According to the data flow, we use 70.51% of the BRAM resources on ZCU 104 to complete this SoC architecture design.

D. PERFORMANCE OF HARDWARE ARCHITECTURE

Tables 8 to Table 9 show the execution times of the proposed hardware architecture on ZCU104 for AlexNet [1], VGG16 [3], and MobileNet-V1 [37] respectively. Here we provide the execution time with two designs, one with and the other without the sparse compression method. Comparing the sparse mode to the dense mode, we add the data sparsity encoder and decoder module before the memory bank to prevent the storage of zero values. These values are not worth further computation since the computation will always result in zero. The remaining values are reordered in the encoder and decoder module and stored inside the memory bank afterward. Thus, the PE array can handle sparse data in the same way as dense data.

We used the ImageNet [1] dataset to train the FC layer network. Here, the batch size is set to 1, and the epoch is set to 1. Due to the nature of the fully connected layer, when the hidden layer is large, many weight parameters need to be transferred from external memory to on-chip memory for computation. If a sparse approach is used, a significant amount of transfer time can be saved. From Table 8 to Table 9, we can see that it takes more time to perform backpropagation. It is mainly because, in backpropagation, we need to calculate the Delta/Error value for each layer, the weight gradient value for each weight, and update each weight using the weight gradient. Tables 8 to 10 shows that the execution time can be reduced by 32.45% to 95.43% using our proposed sparsity approach.

We use the PYNQ framework to complete the overall design as the SoC. In the overall system design, it takes much time to start the DMA where it transfers the data from PS to PL or to transfer the result from PL to PS. We found that it takes most of the execution time to start the DMA and send the data to the ZCU 104. If the sparsity in the network is exploited to reduce the number of weights transferred, the number of initiating DMAs can be reduced and the execution time required for each layer can be reduced. In Tables 8 and 10, the first layer has the largest number of weights so it takes more execution time to execute the forward and reverse propagation of the first layer. In

TABLE 10. Layer by layer execution times for VGG-16 on ZCU104.

Layer	Input size	Filter size	Output node	Parameters	Zero in Ifmap (%)	Time w/o sparse compression (ms)	Time w/ sparse compression (ms)	Reduction ratio	Average throughput (GOPS)
FP-FC1	1024x1x1	1x1	1000	1,024,000	60	17.69	11.95	32.45%	92.4
BP-FC1	1000x1x1	1x1	1024	2,048,000		25.7	14.234	44.62%	89.11
Total						43.39	26.184	39.65%	90.43

TABLE 11. Comparison with other works.

	2016 [5]	2017 [8]	2019 [17]	2021 [10]	2021 [18]	This Work
Platform	Maxeler MPC-X	ZU19EG	Stratix 10	Zynq XC7Z100	ZYNQ ZCU 104	ZYNQ ZCU 104
Frequency	100 MHz	200 MHz	240 MHz	150 MHz	100 MHz	200 MHz
Arithmetic Precision	FP 32	FP 32	Fixed 16	Fixed 16	FP 32	BFP 16
BRAM (36 Kb)	510	174	10.6	679.5	304	220
DSP	23	1500	1699	64	12	1285
FF	87580	466047	-	46690	219372	26832
LUT	69510	329288	20800	40492	169143	73622
Power (W)	27.3	14.24	20.6	2.5	0.67	6.444
Throughput (GOPS)	62.02	86.12	163	19.2	4.39	102.43
Energy Efficiency (GOPS/W)	2.272	6.05	7.9	7.68	6.55	15.895

Table 9, since MobileNet-V1 uses only one layer of the fully-connected network and only 1024 input features, the number of weight parameters is significantly less than AlexNet and VGG16. This eliminates the need to start the DMA on the PS side multiple times to pass the weights, resulting the less execution time.

E. COMPARISON WITH RELATED WORKS

Table 11 shows the comparison of the proposed accelerator with the existing FPGA accelerators in terms of resources, power consumption, and operational performance. The power data is obtained after the FPGA synthesis results, while the throughput information is calculated by multiplying the clock frequency with the number of operations that will be computed in one clock cycle. Due to the different board models of FPGAs, different training accelerators present different performances. For comparison purposes, the evaluation metric is energy efficiency (GOPS/W). Our design can reach 102.4 GOPS at an operating frequency of 200 MHz. Compared with floating-point works in [5], [8], and [18], our design achieves higher energy efficiency. Although the proposed hardware architecture can only handle forward propagation and backward propagation of fully connected layers, our architecture supports softmax and cross-entropy loss functions for the complete training of image classification tasks. The proposed hardware design achieves better performance compared to the works of [10] that supported training and inference of only FC layers. The proposed processor achieves 2.14x higher energy efficiency than [10]. The training accelerator in [17] uses a batch size of 10 to process more image data in parallel, thus reducing the energy consumption per image. However, in [17], they use a batch size of 1 and it leads to higher energy consumption by the additional latency imposed on DRAM with frequent weight updates. With the advantage of high energy-efficient performance, our proposed architecture is more suitable for deployment on mobile devices.

V. CONCLUSION

This paper presents a training processor that performs the training and inference phases of the FC layer. The processor uses a 16-bit brain floating-point computation format to achieve a high-performance hardware design while supporting sparse data. Our proposed hardware architecture can support forward and backward propagation of fully connected layers with a complete training mechanism. We can also reduce the number of DMA reads by expanding the number of PEs and the BRAM usage according to the hardware resources of the development board. The final design is implemented on the Xilinx ZCU104. The synthesis result of 256 MACs can reach 102.4 GOPS at an operating frequency of 200 MHz. We design the architecture in a reconfigurable way to support a different number of nodes, classes, batch size, and the epoch. As long as the parameters in the configuration registers are set at the beginning, end-to-end training is possible. Our architecture also can transfer fully connected layers to other types of layers since we can compute the convolution layers by transferring the convolution operation to matrix multiplication with the im2col method. In the future, the processor will be extended to support 2D convolution and Recurrent Neural Networks to support multiple types of training on a single chip.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inform. Process. Syst.*, 2012, pp. 1097–1105.

[2] J. Redmon, S. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition,", 2015, *arXiv:1409.1556*.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015, *arXiv:1512.03385*.

[5] W. Zhao et al., "F-CNN: An FPGA-based framework for training convolutional neural networks," in *Proc. IEEE 27th Int. Conf. Appl.-Spec. Syst. Architect. Process. (ASAP)*, 2016, pp. 107–114, doi: 10.1109/ASAP.2016.7760779.

[6] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "FPGA acceleration of recurrent neural network based language model," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, 2015, pp. 111–118.

[7] Z. Yuan et al., "STICKER: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, Feb. 2020.

[8] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An FPGA-based processor for training convolutional neural networks," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, 2017, pp. 207–210, doi: 10.1109/FPT.2017.8280142.

[9] D. Han, J. Lee, J. Lee, and H.-J. Yoo, "A low-power deep neural network Online learning processor for real-time object tracking application," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 5, pp. 1794–1804, May 2019.

[10] X. Chen, C. Gao, T. Delbrück, and S.-C. Liu, "EILE: Efficient incremental learning on the edge," in *Proc. IEEE 3rd Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2021, pp. 1–4.

[11] K. Guo et al., "Compressed CNN training with FPGA-based accelerator," in *Proc. 2019 ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2019, p. 189.

[12] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of CNN training on deeply-pipelined FPGA clusters with work and weight load balancing," in *Proc. 28th Int. Conf. Field Program. Logic and Appl. (FPL)*, 2018, pp. 394–398, doi: 10.1109/FPL.2018.00074.

[13] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[14] S. Han et al., "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architect. (ISCA)*, 2016, pp. 243–254.

[15] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," 2018, *arXiv:1804.07612*.

[16] M. A. Hussain and T.-H. Tsai, "Memory access optimization for on-chip transfer learning," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 4, pp. 1507–1519, Apr. 2021.

[17] S. K. Venkataramanaiah et al., "Automatic compiler based FPGA accelerator for CNN training," in *Proc. 29th Int. Conf. Field Program. Logic Appl. (FPL)*, 2019, pp. 166–172, doi: 10.1109/FPL.2019.00034.

[18] J. Hong, S. Arslan, T. Lee, and H. Kim, "Design of power-efficient training accelerator for convolution neural networks," *Electronics*, vol. 10, no. 7, p. 787, 2021. [Online]. Available: https://doi.org/10.3390/electronics10070787

[19] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2019, pp. 151–165.

[20] E. Qin et al., "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect. (HPCA)*, 2020, pp. 58–70.

[21] A. Aimar et al., "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019, doi: 10.1109/TNNLS.2018.2852335.

[22] T. Yuan, W. Liu, J. Han, and F. Lombardi, "High performance CNN accelerators based on hardware and algorithm co-optimization," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 1, pp. 250–263, Jan. 2021, doi: 10.1109/TCSI.2020.3030663.

[23] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010, doi: 10.1145/1816038.1815993.

[24] Y. Umuroglu et al., "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 65–74, doi: 10.1145/3020078.3021744.

[25] K. Guo et al., "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018,

[26] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An FPGA-based CNN accelerator integrating depthwise separable convolution," *Electronics*, vol. 8, no. 3, p. 281, 2019.

[27] H. Shao, J. Lu, J. Lin, and Z. Wang, "An FPGA-based reconfigurable accelerator for low-bit DNN training," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2021, pp. 254–259, doi: 10.1109/ISVLSI51109.2021.00054.

[28] "IEEE 754." Accessed: Jun. 2022. [Online]. Available: https://zh.wikipedia.org/wiki/IEEE_754

[29] A. Agrawal et al., "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *Proc. IEEE 26th Symp. Comput. Arithmet. (ARITH)*, 2019, pp. 92–95, doi: 10.1109/ARITH.2019.00023.

[30] U. Köster et al., "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proc. NIPS*, 2017, pp. 1742–1752.

[31] "TF32." NVIDIA, Santa Clara, CA, USA. Accessed: Jun. 2022. [Online]. Available: https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/

[32] W. Shibo and K. Pankaj. "BFloat16: The secret to high performance on cloud TPUs." Aug. 2019. [Online]. Available: https://reurl.cc/43Z7qY

[33] C. Chen, H. Ding, H. Peng, H. Zhu, Y. Wang, and C. J. R. Shi, "OCEAN: An on-chip incremental-learning enhanced artificial neural network processor with multiple gated-recurrent-unit accelerators," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 8, no. 3, pp. 519–530, Sep. 2018, doi: 10.1109/JETCAS.2018.2852780.

[34] C.-H. Lu, Y.-C. Wu, and C.-H. Yang, "A 2.25 TOPS/W fully-integrated deep CNN learning processor with on-chip training," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, 2019, pp. 65–68.

[35] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Comput.*, vol. 11, no. 4, pp. 853–862, May 1999.

[36] C. S. Turner, "A fast binary logarithm algorithm [DSP tips & tricks]," *IEEE Signal Process. Mag.*, vol. 27, no. 5, pp. 124–140, Sep. 2010, doi: 10.1109/MSP.2010.937503.

[37] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications." 2017. [Online]. Available: https://arxiv.org/abs/1704.04861.