electronics

MDPI

Article

FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit

Mannhee Cho [1] and Youngmin Kim [2,*]

[1] School of Electrical Engineering, Korea University, Seoul 02841, Korea; mhc9840@gmail.com
[2] School of Electronic and Electrical Engineering, Hongik University, Seoul 04066, Korea
* Correspondence: youngmin@hongik.ac.kr; Tel.: +82-2-320-1665

Abstract: Convolutional neural networks (CNNs) are widely used in modern applications for their versatility and high classification accuracy. Field-programmable gate arrays (FPGAs) are considered to be suitable platforms for CNNs based on their high performance, rapid development, and reconfigurability. Although many studies have proposed methods for implementing high-performance CNN accelerators on FPGAs using optimized data types and algorithm transformations, accelerators can be optimized further by investigating more efficient uses of FPGA resources. In this paper, we propose an FPGA-based CNN accelerator using multiple approximate accumulation units based on a fixed-point data type. We implemented the LeNet-5 CNN architecture, which performs classification of handwritten digits using the MNIST handwritten digit dataset. The proposed accelerator was implemented, using a high-level synthesis tool on a Xilinx FPGA. The proposed accelerator applies an optimized fixed-point data type and loop parallelization to improve performance. Approximate operation units are implemented using FPGA logic resources instead of high-precision digital signal processing (DSP) blocks, which are inefficient for low-precision data. Our accelerator model achieves 66% less memory usage and approximately 50% reduced network latency, compared to a floating point design and its resource utilization is optimized to use 78% fewer DSP blocks, compared to general fixed-point designs.

Keywords: convolutional neural network; FPGA; high-level synthesis; accelerator

Citation: Cho, M.; Kim, Y. FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit. *Electronics* **2021**, *10*, 2859. https://doi.org/10.3390/electronics10222859

Academic Editors: Sotiris Ioannidis, Konstantinos Georgopoulos, Iakovos Mavroidis and Jose Santamaria

Received: 15 September 2021
Accepted: 17 November 2021
Published: 19 November 2021

1. Introduction

In many modern applications, convolutional neural networks (CNNs) are adopted for image classification based on their high versatility and accuracy. Many recent studies have proposed novel CNN architectures, application systems, and optimization methods for both software and hardware platforms [1–8].

Optimizations for CNN accelerators generally involve exploring optimized data types and network architectures. Traditional CNNs use high-precision floating-point data types for both training and inference, but many recent studies have explored more efficient data types by reducing data sizes and applying quantization [9–14]. Many studies have proven that CNNs can achieve improvements in performance and resource utilization by using low-precision data without a significant loss of classification accuracy. Network architectures can be optimized to increase memory access bandwidth, efficient resource utilization, and parallelization for improved latency. Many network architectures have been explored and developed with various trade-offs between resources.

Many studies on CNN acceleration have adopted field-programmable gate arrays (FPGAs) as hardware platforms for evaluating performance because FPGAs have the advantages of reasonably high performance, rapid development, and reconfigurability, using software tools [15–29]. They also have built-in complex computational units, such as digital signal processing (DSP) blocks for implementing large-scale arithmetic operations with maximum performance [30]. CNNs are limited by their large memory size, computational

resources, and power consumption. To achieve high inference speed, additional resources and power are required as trade-offs. However, newly developed FPGAs contain a large number of computational units and optimized power systems that are sufficient for implementing high-speed and high-power CNN accelerator models, which are suitable for implementing and experimenting with various CNN models.

However, FPGAs are large arrays of pre-constructed hardware-mapped elements. The logic and functions designed by users are synthesized to fit into these elements and may fail to utilize them optimally. In particular, DSP blocks perform arithmetic operations with high precision and are underutilized when fed low-precision data. Increasing the efficiency of FPGA resources is also a challenge in CNN accelerator design [15,21–23]. We can implement more efficient arithmetic operators in place of DSP blocks when implementing CNN accelerators for more optimized resource utilization and management.

The software tools provided by FPGA manufacturers are used when developing hardware accelerators for CNNs on FPGA platforms. Recently, various high-level synthesis (HLS) tools were developed. HLS tools can automatically synthesize register-transfer-level (RTL) designs from source code written in high-level languages (e.g., C/C++). For Xilinx FPGAs, the Vivado and Vitis HLS tools were developed. HLS tools can support various data types, including both floating- and fixed-point data types with arbitrary bit lengths, as well as pragma directives for timing, resource configuration, and the software verification of designs. Bitwise operations can also be synthesized, facilitating precise hardware-optimized operations. Many studies were conducted by using HLS tools to develop and implement CNN accelerator models on FPGAs [15,18,19,26,27,29].

In this paper, we propose a hardware accelerator design for the LeNet-5 CNN architecture [31], which is a CNN architecture for handwritten digit classification that was trained and tested on the MNIST handwritten digit dataset [32]. We implemented the proposed accelerator on a Xilinx XCZU9EG-2ffvb1156 FPGA chip, using the Xilinx Vitis HLS tool (v2020.2). A set of 10,000 MNIST handwritten digit images were used for inference to evaluate accuracy. As a baseline, we applied loop optimizations and a network dataflow scheme to improve computation parallelization and reduce inference latency. To further optimize the network, we adopted a fixed-point data type supported by the HLS tool. Based on the tanh activation function used for the LeNet-5 architecture, we optimized the data transferred to the memory by removing unused integer bits after activation, improving memory efficiency. We also implemented approximate multiply-accumulate (MAC) units using bit-level data modification to reduce computational cost. This also reduces the number of DSP blocks used, which are limited in numbers in FPGAs. We experimented with multiple designs using various data types and sizes to analyze and compare the results on performance and resource utilization.

The contributions of this study are as follows:

- An approximate MAC operator based on bit modifications and functions provided by HLS tool is proposed and implemented for CNN accelerator on FPGA.
- Additional data size optimization for CNN is applied by removing unused bits after output activation function processing.
- Experiments were performed with various bit width for data on HLS implementation of CNN accelerator, and the performance results are analyzed.

The remainder of this paper is organized as follows: Section 2 presents background information on CNNs, the LeNet-5 CNN architecture, and fixed/floating-point data types. Section 3 describes the proposed CNN accelerator design and the applied optimization methods in detail. Section 4 analyzes the experimental results, followed by conclusions in Section 5.

2. Background

2.1. CNN

A CNN is a type of deep neural network (DNN) that utilizes a convolution algorithm based on a 2D array of inputs. Although the output is the sum of multiplied inputs and weights, similar to a traditional DNN, a CNN uses kernels. Kernels are groups of weights that perform sliding-window convolution operations on input feature maps. Each output node is the sum of overlapping input feature maps and kernels. This is also known as a shared-weight scheme. This allows a CNN to reduce the number of trainable parameters significantly while accelerating network training and inference.

To reduce network size and computations, pooling layers, which are also called sub-sampling layers, can be adopted. Pooling layers aim to reduce input dimensions by extracting meaningful data from each region of an input. A kernel, which represents the batch of units to be computed or compared together, can use varying sizes and stride values to generate different output dimensions. Depending on the network, pooling layers can also have trainable parameters, such as weight and bias values. In general, average pooling and max (or min) pooling methods are used. Average pooling computes the average of the surrounding units, whereas max pooling compares values and outputs the maximum value. Average pooling is more computationally expensive than max pooling, but partially preserves all input values by calculating the average of a batch. Max pooling drops all values, except for the maximum value, but is computationally cheaper. The optimal method depends on the target network and data. One method may yield better results than the other for a given network [33–35].

Activation functions are used to introduce nonlinearity into networks. Activation functions process the output values of nodes before they are passed to the next layer. Without an activation function, a neural network is essentially a linear regression model. As shown in Figure 1, complex classification cannot be achieved by linear models. Therefore, nonlinear activation functions are used to construct more complex models. There are multiple types of activation functions. Some examples are presented in Figure 2. In addition to differences in arithmetic complexity, activation functions can also affect network training and accuracy. Therefore, identifying the optimal activation function for a target network is important [36,37].

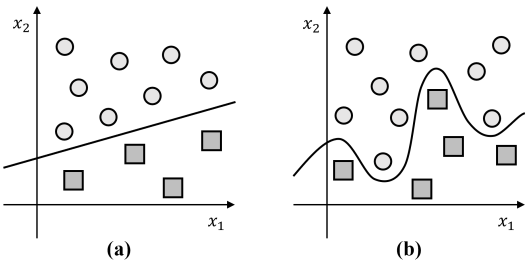


Figure 1. (**a**) Linear model and (**b**) nonlinear model.

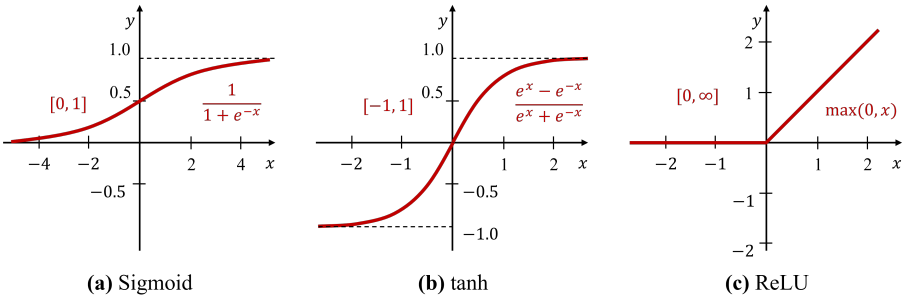


Figure 2. Example activation functions: (**a**) sigmoid, (**b**) tanh, and (**c**) ReLU.

2.2. LeNet-5

LeNet-5 is a CNN architecture developed by LeCun et al. [31]. It is a handwritten digit classifier architecture that was trained and tested on the MNIST handwritten digit dataset [32]. The network architecture is presented in Figure 3 and the data flow is presented in Figure 4. It consists of three convolutional layers (C1, C3, and C5), two pooling layers (S2 and S4), and two fully connected layers (F6, OUTPUT). Each layer uses a tanh activation function. The input is a 32 × 32 handwritten digit image, and the outputs are digit classification results (zero to nine). The layer configuration of LeNet-5 is presented in Table 1.

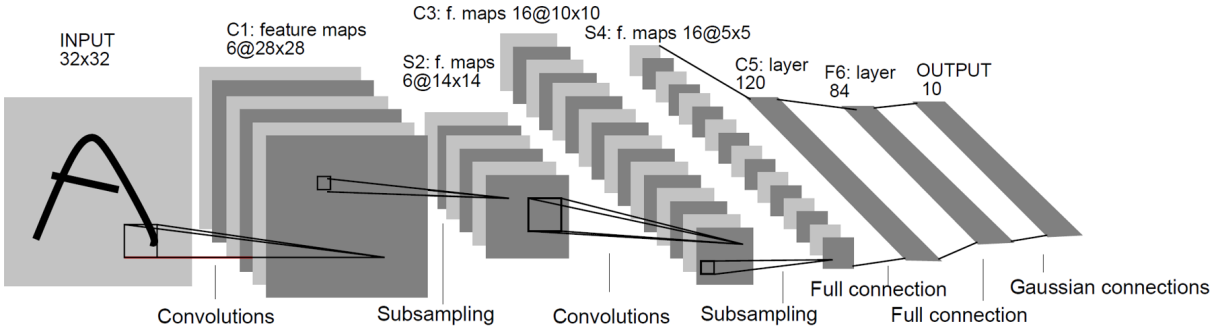


Figure 3. LeNet-5 CNN architecture [31].

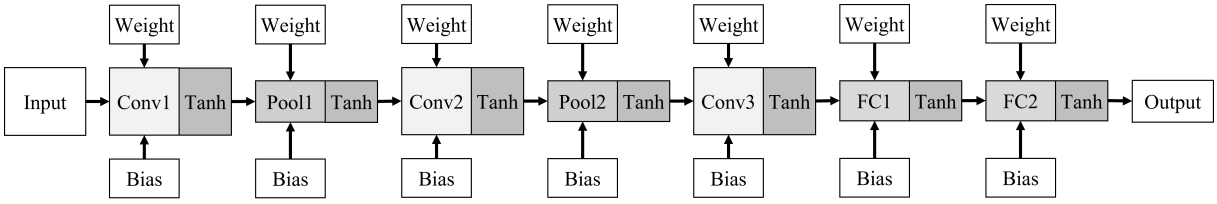


Figure 4. Structural data flow of the LeNet-5 CNN architecture [31].

Table 1. Layer configuration of LeNet-5 [31].

Layer	Input	Weight	Bias	Kernel	Stride	Output
Conv1	1 × 32 × 32	6 × 1 × 5 × 5	6	5	1	6 × 28 × 28
Pool1	6 × 28 × 28	6	6	2	2	6 × 14 × 14
Conv2	6 × 14 × 14	16 × 6 × 5 × 5	16	5	1	16 × 10 × 10
Pool2	16 × 10 × 10	16	16	2	2	16 × 5 × 5
Conv3	16 × 5 × 5	120 × 16 × 5 × 5	120	5	1	1 × 120
FC1	1 × 120	120 × 84	84	-	-	1 × 84
FC2	1 × 84	84 × 10	10	-	-	1 × 10

The convolutional layers perform convolutions, using a 5 × 5 kernel with a stride of one. The number of trainable weights is the product of the number of input feature maps, output feature maps, and kernel size (5 × 5). The number of biases is the same as the number of output feature maps. Pooling layers perform 2 × 2 average pooling operations with a stride of two, reducing the input image's width and height by half. LeNet-5 also uses trained weights and biases. As shown in Figure 5, after average processing, feature maps are multiplied by weights and then added to biases. This results in trained weights and biases of the same number as the input feature maps.

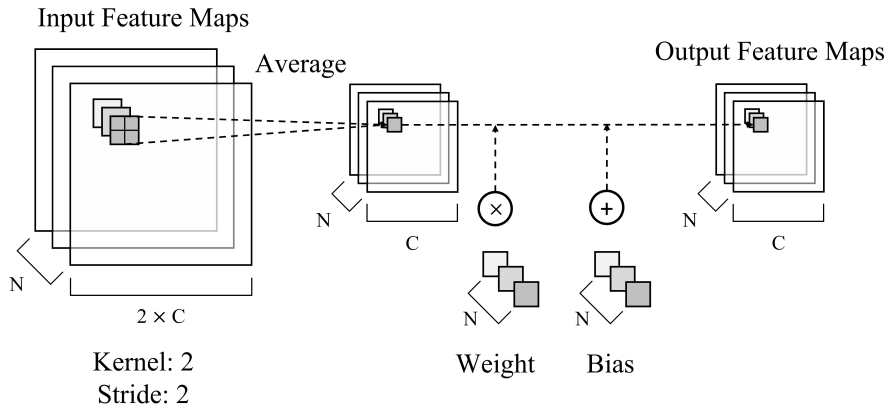


Figure 5. Average pooling with weights and biases.

The fully connected layers have the same structure as traditional artificial neural network layers. All inputs are connected to all outputs. The number of weights is the product of the numbers of input and output neurons. The number of biases is the same as the number of output neurons.

2.3. FPGA

An FPGA is a device composed of configurable logic blocks and other circuitry, such as memory blocks, which are connected by path-switching circuits called programmable interconnects. An FPGA can be programmed to perform functions designed by users and can be reprogrammed at any time. Although FPGAs may provide lower performance than application-specific integrated circuits, the reconfigurability of FPGAs can significantly reduce development time, so they are often selected as hardware platforms for various experiments. Recently developed FPGAs also contain high-performance clocking resources and a large number of processing units, such as DSP blocks, allowing users to develop high-speed applications (including CNNs) with ease. FPGAs are programmed using software tools provided by FPGA manufacturers that generally use a hardware description language. However, recently developed HLS tools allow users to write system code in high-level languages, which is then synthesized to the register transfer level. Such tools also provide various functions, such as pragma directives that can automatically configure resource utilization limits and timing constraints for synthesis.

Despite the excellent performance of FPGAs, because logic blocks are built-in units with fixed sizes, it is possible that in practice, resources may not be utilized at their peak performance. MAC operations used in computationally heavy systems, such as CNN accelerators, are generally implemented using DSP blocks. An example of an FPGA DSP block (Xilinx DSP48E block [30]) is presented in Figure 6. This block contains a precise 27×18 bit multiplier, which is less efficient when using low-precision data. Additionally, the number of DSP blocks is limited in an FPGA, so a non-optimized algorithm can result in limited throughput and require a larger FPGA chip. This not only applies to DSP blocks, but also to other logic elements. In the worst cases, a small change in the data bit width can significantly change how an algorithm is implemented, thereby affecting the performance and resource utilization of the entire system. With the wide use of FPGAs in modern applications, many studies have focused on the efficient implementation of systems on FPGAs. Several studies on FPGA CNN accelerators have performed algorithm modification for resource usage management and optimized data types for logic elements [21–23].

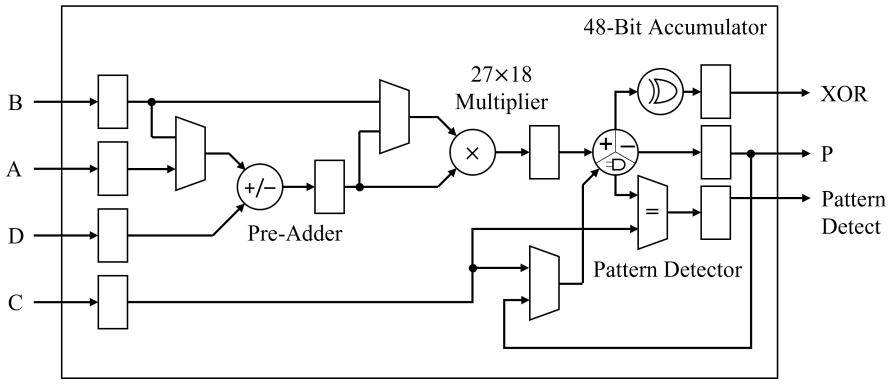


Figure 6. Xilinx FPGA DSP48E2 block [30].

3. Proposed Accelerator Design

The proposed accelerator is based on the LeNet-5 architecture. We used the Xilinx Vitis HLS (v2020.2) tools to synthesize the accelerator design targeting the Xilinx XCZU9EG-2ffvb1156 FPGA chip. The accelerator models are implemented on programmable logic cells of the FPGA chip without using MPSoC processor cores, and thus can be implemented on any other FPGA chips. The accelerator receives input data in a 32-bit floating-point data type and converts them into fixed points. The fixed-point outputs of the network are converted into 32 bit floating-point values and exported as the final outputs. The network performs the initial setup once, and then the stored weights and biases in the internal memory can be reused for new sequences of input image sets.

The proposed CNN accelerator utilizes three major optimizations: loop parallelization, fixed-point data optimization, and approximate MAC operations. The details of these optimizations are explained in the following subsections.

3.1. Loop Parallelization

Loop parallelization is achieved by using HLS pragma directives provided by the Vitis HLS tools. "#pragma HLS Unroll" is used to flatten loops. When synthesized, operations in the loop body are implemented as multiple instances that operate in parallel, as shown in Figure 7, which significantly reduces latency at the cost of additional computational resources. Additional optimization is performed by using "#pragma HLS Pipeline" to divide operations in small stages for concurrent execution, as shown in Figure 8. The pseudocode for a convolutional layer with pragma directives is presented in Figure 9.

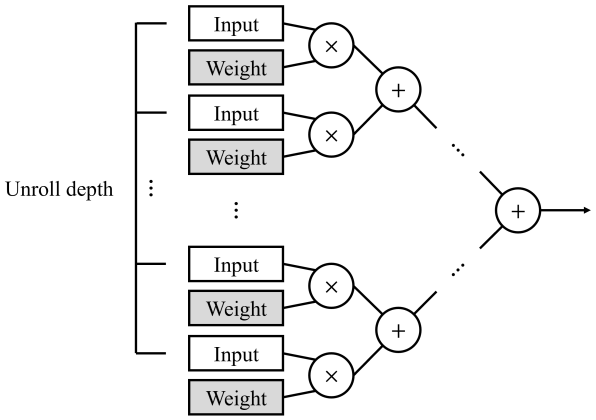


Figure 7. Unrolled arithmetic units.

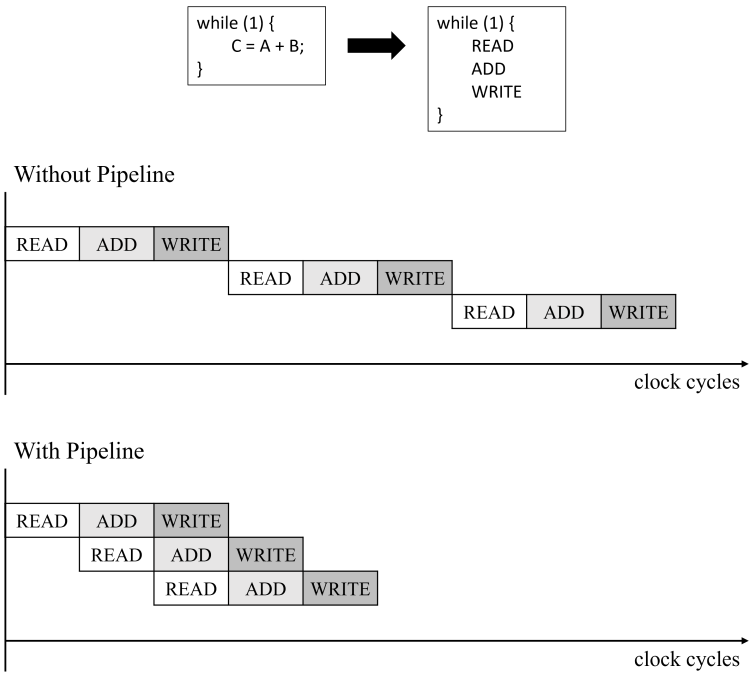


Figure 8. Pipelined operations.

```
for (depth_in = 0; depth_in < N; depth_in++) {

   for (row = 0; row < R; row++) {
   for (col = 0; col < C; col++) {
#pragma HLS pipeline

      for (row_k = 0; row_k < K; row_k++) {
#pragma HLS unroll
      for (col_k = 0; col_k < K; col_k++) {
#pragma HLS unroll

         for (depth_out = 0; depth_out < M; depth_out++) {
#pragma HLS unroll
            Output[] += Input[] * Kernel[]
         }
      } }
   } }
}
```

Figure 9. Pseudocode for a convolutional layer with pragma directives.

The entire network is pipelined using "#pragma HLS Dataflow". This directive allows functions to operate in a pipelined manner, increasing the throughput of the accelerator. As shown in Figure 10, because the input of each layer is dependent on the output of the previous layers, all layers must be executed in order. Executing multiple layers in parallel for the same network input is not possible. Additionally, each layer must finish processing its current input (generate an output) before accepting another set of inputs to prevent the internal registers from being overwritten during its operations. Therefore, the minimum number of clock cycles required for the accelerator to accept the next input image is the same as the number of clock cycles (plus one) of the layer requiring the greatest number of cycles.

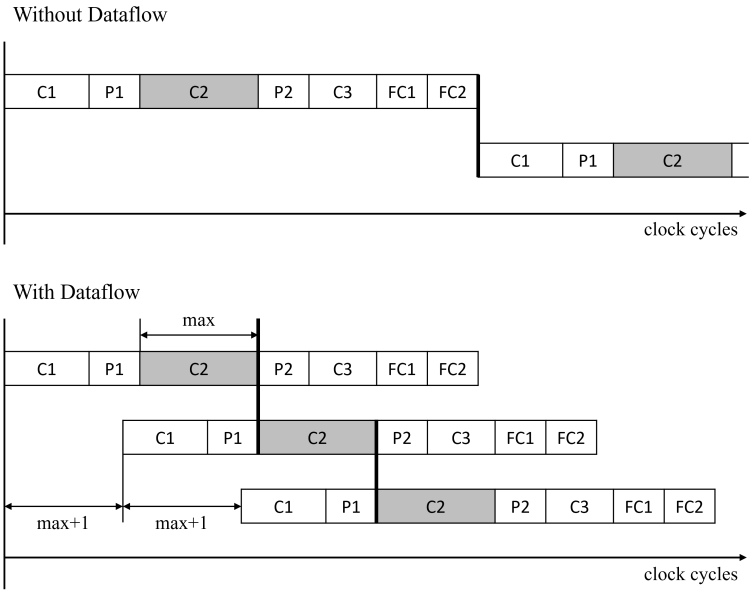


Figure 10. Pipelined network using dataflow directives.

3.2. Fixed-Point Data Optimization

Floating-point and fixed-point formats are two major representations of real numbers in computing. The structures of these two formats for the same 32 bit length are presented in Figure 11. The IEEE-754 single-precision binary floating-point format is composed of 1 sign bit, 8 exponent bits, and 23 significant bits for a total of 32 bits. Data are represented by their significance and scaled by the exponent as a power of two. The term "float" refers to the fact that the decimal point in the number can move relative to the significant digits. Therefore, the floating-point format can represent a wide range of numbers.

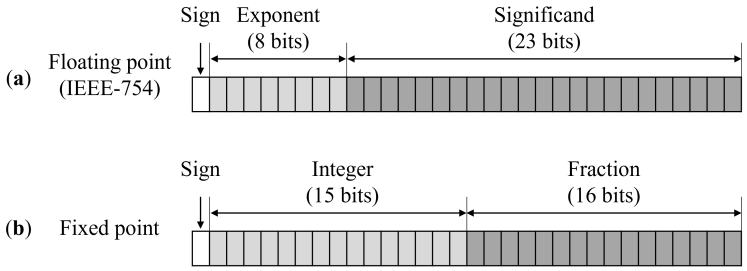


Figure 11. The 32 bit (**a**) floating-point format and (**b**) fixed-point format.

The fixed-point format consists of integer bits (including sign bits) and fractional bits. Fixed-point data are essentially binary data shifted by a given static factor. The positions of the bits are fixed without additional digit shifting, unlike floating-point data. Although the precision of data is limited by the number of bits, fixed-point arithmetic is more computationally efficient. For hardware implementations, using a fixed-point data type can reduce the area, power, and latency of arithmetic processing units.

CNNs are typically implemented and trained on GPUs and CPUs, using a floating-point data type for high precision. However, many recent studies have proposed a method of using a fixed-point data type for both training and inference acceleration [9,16,18,19,21]. Although fixed-point data have limited precision and result in the gradual loss of data, recent research has shown that using fixed-point data for CNNs can yield approximately the same results as using floating-point data when given a sufficient number of bits. The trade-off between network accuracy and performance (i.e., area, power, and latency) is important, so a smaller length of fixed-point data can be adopted in some scenarios. For FPGA designs, using fixed-point data can reduce resource usage, power, and latency [38].

To determine the optimal number of fractional bits, we can test experimental models with different data sizes. The number of integer bits is dependent on the network architecture, particularly the size of the convolution kernels. When the accumulated results from MAC operations on inputs and weights exceed the number of integer bits available for memory, overflow may occur, leading to critical errors in a network. We found that in our CNN architecture, the minimum number of integer bits required to prevent overflow during accumulation is six bits (including one sign bit). This allows us to store values ranging from -32 to 31, which is sufficient for a 5×5 kernel with multiplied values ranging from -1 to 1.

Because the LeNet-5 network uses the tanh activation function to process the output of each layer, the size of each layer output ranges from -1 to 1. Therefore, we can further reduce the memory size for storing data and port width between layers by truncating unused integer bits. As shown in Figure 12, the 12 bit data resulting from MAC operations can be truncated by four integer bits, yielding 8 bit data. This can also improve the memory access throughput.

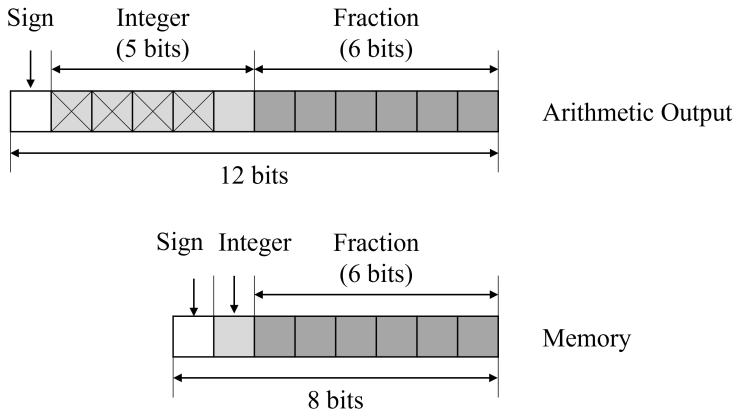
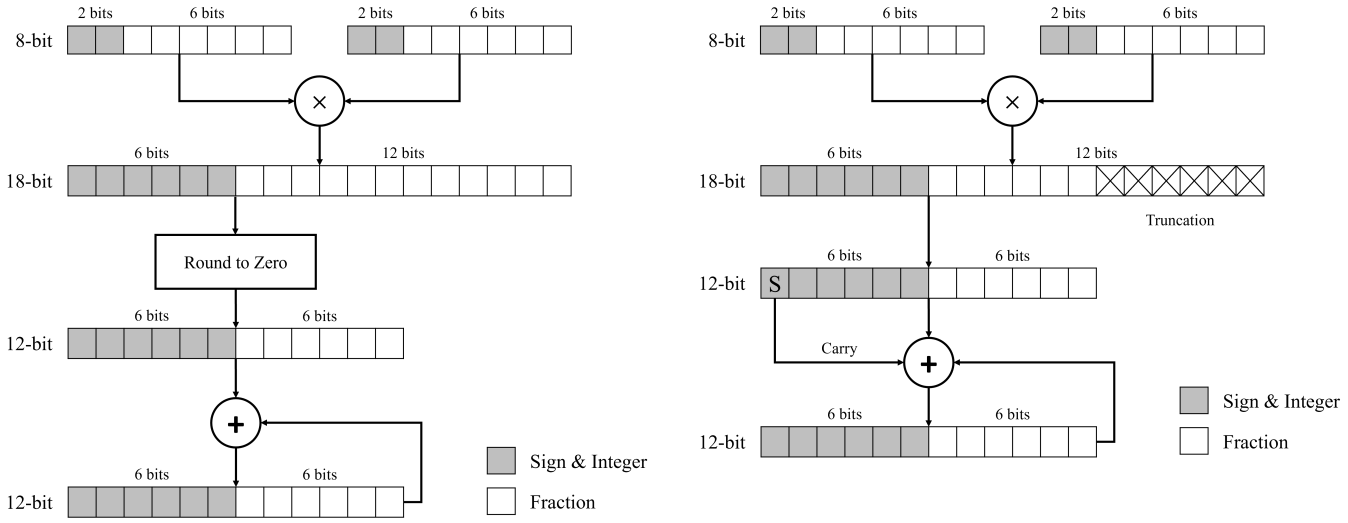


Figure 12. Removing unused integer bits from signed fixed-point data.

The proposed accelerator uses low-precision signed fixed-point data with two integer bits and six fractional bits to store parameter values. This has the benefits of significantly reducing memory size and resource usage with little loss in accuracy when using our approximate MAC operation units for convolutional layers, which are described in Section 3.3. For pooling and fully connected layers, the accelerator uses 12 bit fixed-point operations. Operation units are synthesized as lookup tables (LUTs) and flip-flops instead of DSP blocks. Data are truncated when the result of multiplication exceeds 12 bits in length (i.e., one sign bit, five integer bits, and six fractional bits). Our experiments revealed that the precision of the pooling and fully connected layers has less impact on the classification accuracy, compared to that of the convolutional layers.

3.3. Approximate MAC Operations

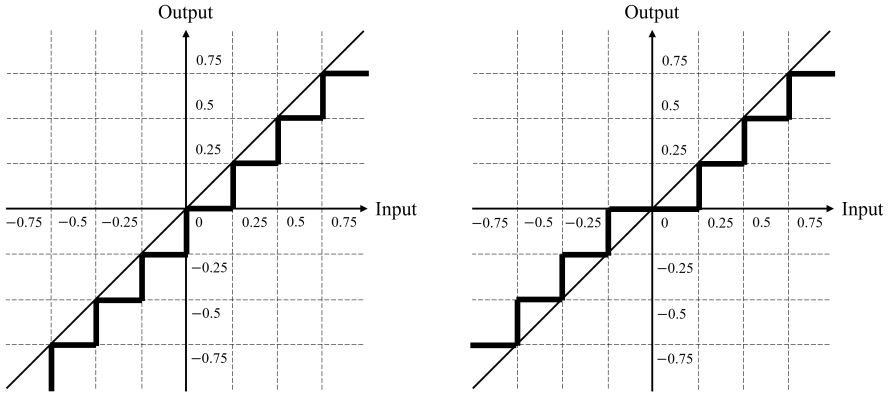
Our accelerator design contains two models using two different MAC operator modules for the convolutional layers. Figure 13a presents the rounded MAC module, and Figure 13b presents the carry MAC module. Multiplication is performed with 18 bit precision, but when passing the data to the accumulator, rounding or a carry bit is applied to remove the lower six fractional bits. These methods reduce the complexity of the accumulation stage without having a heavy impact on the arithmetic results. The proposed MAC operators are implemented on FPGAs in the form of LUTs and logic blocks instead of using high-precision DSP blocks, which would be inefficient based on the small data size.



(a) Rounded MAC **(b)** Carry MAC

Figure 13. Proposed approximate MAC units: (**a**) rounded MAC and (**b**) carry MAC.

The rounded MAC module uses a round-to-zero quantization method for the fixed-point data type. As shown in Figure 14, the truncation of fixed-point data results in data rounded to the floor (minus infinity). Our experiments revealed that simply truncating the lower six fractional bits results in critically reduced classification accuracy. Therefore, a round-to-zero operation is applied to make the rounded value symmetrical for positive and negative values. This process is automatically synthesized as a hardware function, as defined by the HLS tools.



(a) Truncation (b) Round to zero

$$
\begin{aligned}
-2.625 &= 1\,|\,101\,|\,01\underline{10} \\
-2.75 &= 1\,|\,101\,|\,01 \qquad \text{(Truncation)} \\
-2.5 &= 1\,|\,101\,|\,10 \qquad \text{(Round to zero)}
\end{aligned}
$$

Figure 14. Comparison of truncation and round-to-zero operations for fixed-point data.

The carry MAC module passes an extra carry bit to the adder instead of performing the rounding function. The lower six bits are completely truncated. The carry bit is the sign bit of the resulting signed fixed-point data. Because negative values are represented as two's complement, truncating the lower bits rounds the value to minus infinity, in contrast to positive values, which are rounded to zero. The added carry bit pulls a negative value toward positive infinity.

4. Experimental Results

The proposed and experimental accelerator models were designed using the Vitis HLS tool (v2020.2). The number of clock cycles needed for output generation is acquired from synthesis reports on the HLS tool. The accelerator models are then exported to the Xilinx Vivado Design Suite tool (v2020.2) for more accurate analysis. The results on the maximum operating frequency, resource utilization, and power consumption are derived from post-implementation (placement and routing) reports and the power estimation report taken from the Vivado tool.

The network was trained with MNIST handwritten digit dataset training set images on CPU, using the floating-point data type. Inference accuracy results were acquired from the testbench simulation on the HLS tool. A total of 10,000 MNIST test set images were used to obtain the classification accuracy for each model. The trained floating-point parameters are sent to the accelerator models, where they are internally converted into fixed-point data type using hardware functions. No additional training was performed after data optimizations.

We first compared the classification accuracies between various data types. Table 2 presents comparisons of classification accuracy between the 32 bit floating-point model, fixed-point models with various bit lengths, and the proposed accelerators using rounded MAC and carry MAC operations. Only loop parallelization was applied to the floating-point and fixed-point models. Our proposed model applies all three of the optimizations described in Section 3. The fixed-point models use fixed-point data with six integer bits (including sign bits) and different numbers of fractional bits ranging from 6 to 12. The models using fixed-point data with 9 to 12 fractional bits exhibit less than a 1% loss in accuracy. However, using seven bits yielded a notable 10% loss, and using six bits resulted in a very poor accuracy of 34%, both of which are unacceptable. The proposed model with rounding MAC had less than 1% loss, and the carry MAC model had an accuracy loss of approximately 2%, which is acceptable.

Table 2. Comparison of classification accuracies between accelerator models.

Data Type <Integer, Fraction>	Accuracy (10,000 Sets)
Floating (32-bit)	9863
Fixed <6, 12>	9859
Fixed <6, 11>	9858
Fixed <6, 10>	9847
Fixed <6, 9>	9834
Fixed <6, 8>	9758
Fixed <6, 7>	8977
Fixed <6, 6>	3408
Rounded MAC	9821
Carry MAC	9614

Additionally, we experimented with varying fixed-point data precision levels for each layer type. A model with 18 bit data (six integer bits and 12 fractional bits) for the convolutional layers and 12 bit data (six integer bits and six fractional bits) for the pooling and fully connected layers yielded 9840 correct classifications out of 10,000 images, as shown in Table 3. This is very similar to the 9859 correct results for the full 18 bit fixed-point model. Therefore, we concluded that the pooling layers and fully connected layers have very little negative impact on network classification accuracy when using low-precision data.

Table 3. Classification accuracies with varying data precision.

Data Type	Accuracy (10,000 Sets)
18-bit	9859
18/12-bit	9840
12-bit	3408

Next, we compared the timing and resource utilization of each model. The results were obtained from the resource usage and final timing reports outputted by the Xilinx Vivado Design Suite (v2020.2) platform using "export RTL" option on the Vitis HLS tools. The results are based on post-implementation (placement and routing). The timing comparisons are presented in Table 4 and resource utilization is presented in Table 5. Graphs showing both types of results are presented in Figure 15. In Table 4, the clock period is the minimum length of the clock period required for the model to operate correctly. Based on this period, we can obtain the maximum operating frequency. The clock cycles column contains the number of clock cycles between the start of the input data stream and the end of the final output stream. Latency is the product of the minimum clock period and number of clock cycles.

Table 4. Timing comparisons between accelerator models.

Data Type <Integer, Fraction>	Clock Period (ns)	Maximum Frequency (MHz)	Clock Cycles	Latency (ms)	Normalized Latency
Floating (32-bit)	9.428	106.07	1,076,296	10.147	100%
Fixed <6, 12>	7.342	136.20	701,300	5.149	51%
Fixed <6, 11>	7.160	139.66	701,300	5.021	49%
Fixed <6, 10>	7.534	132.73	701,300	5.284	52%
Fixed <6, 9>	7.455	134.14	701,300	5.228	52%
Fixed <6, 8>	7.466	133.94	701,290	5.236	52%
Fixed <6, 7>	8.436	118.54	701,300	5.916	58%
Fixed <6, 6>	7.549	132.47	701,300	5.127	51%
Rounded MAC	8.734	114.50	587,004	5.127	51%
Carry MAC	7.890	126.74	587,004	4.631	46%

Table 5. Comparisons of resource usage between accelerator models.

Data Type <Integer, Fraction>	CLB	LUT	FF	DSP	BRAM	SRL	Latch
Floating (32-bit)	12,405	52,406	46,114	199	303	589	0
Fixed <6, 12>	9653	43,929	31,100	549	159	581	32
Fixed <6, 11>	9256	42,807	29,618	569	165	581	32
Fixed <6, 10>	9236	41,841	28,720	569	163	581	32
Fixed <6, 9>	9245	40,717	28,016	559	150	581	32
Fixed <6, 8>	8932	39,360	27,531	539	144	581	32
Fixed <6, 7>	8661	38,448	26,372	549	128	581	32
Fixed <6, 6>	8030	37,598	25,196	559	124	581	32
Rounded MAC	11,273	61,713	27,863	123	102	545	32
Carry MAC	10,991	57,657	28,311	123	102	581	32

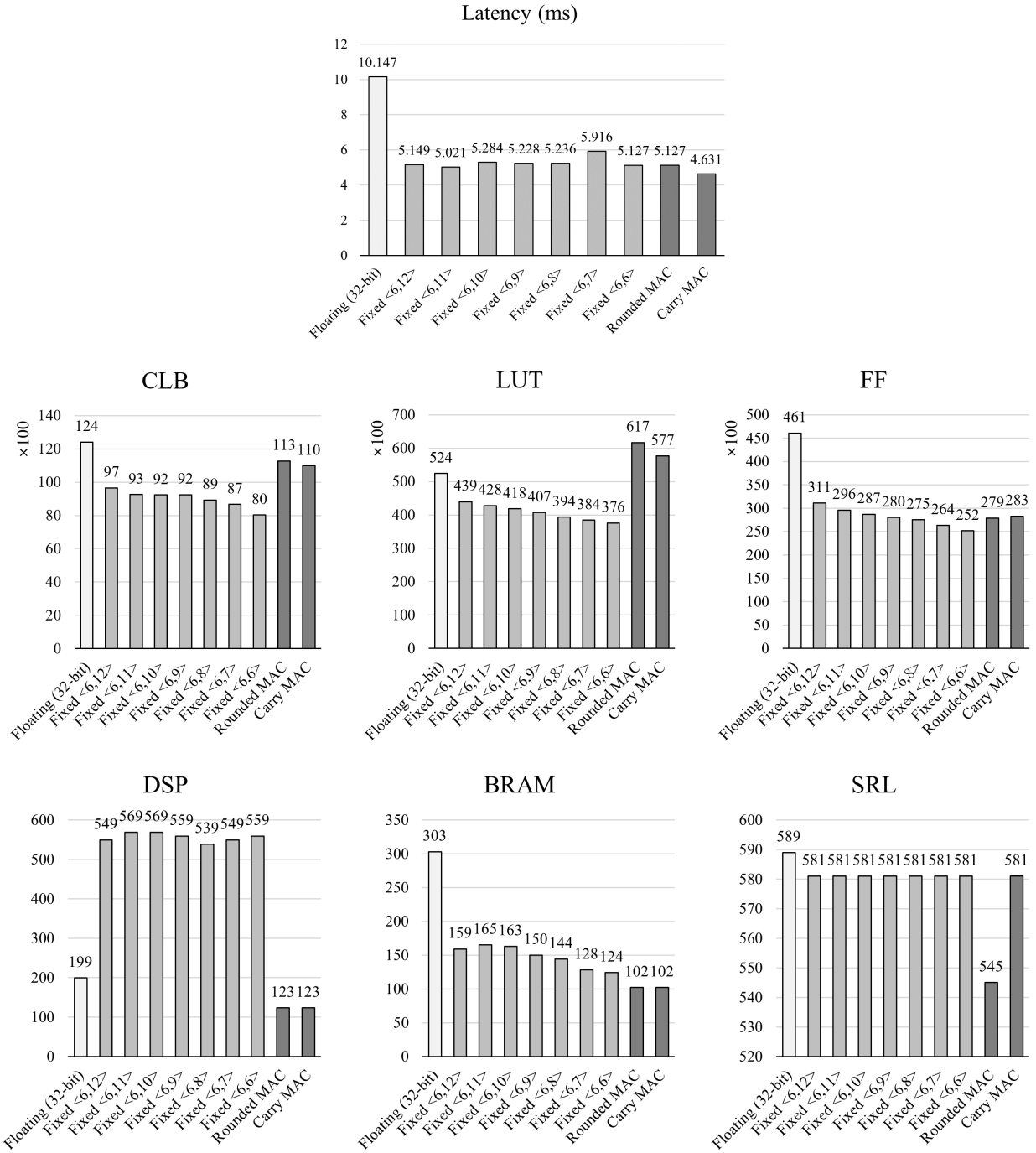


Figure 15. Comparisons of latency and resource usage between accelerator models.

According to Table 4, one can see that the floating-point model has a higher clock period and more clock cycles than the fixed-point model, resulting in approximately twice as much latency. However, when considering both Tables 4 and 5, one can see that timing and resource utilization are not completely proportional to the length of the data type. Although the floating-point model has high latency, it uses fewer DSP blocks compared to the fixed-point models. Among the fixed-point models, the 18 bit model uses fewer DSP blocks than the other models, except for the 14 bit model, and the 12 bit model uses 10 more DSP blocks than the 13 bit model. Regarding the timing, the clock period is not proportional to the length of the data. Based on this result, we can infer that the HLS tool applies algorithm modification and a resource reuse scheme when required or possible. As

a result of the fixed sizes of FPGA elements (such as DSP blocks), changes in the bit length of the input and output can result in significant changes in resource usage and mapping. The floating-point model appears to use fewer DSP blocks by reusing arithmetic units in multiple layers, whereas the fixed-point models use additional DSP blocks to accelerate the network further. Additionally, because the FPGA elements are mapped to fixed locations, changes in resource usage affect the signal paths, thereby changing the maximum operating frequency through critical paths.

Next, we analyze the results of the proposed accelerator models. The rounded MAC model and carry MAC model both yield a reduced number of clock cycles compared to the other models (i.e., 45% less than the floating-point model and 16% less than the fixed-point models). Regarding each type of network layer, while the convolutional and pooling layers have the same numbers of cycles as those in the fixed-point models, the fully connected layer exhibits a noteworthy reduction in cycles. However, our proposed models also have a higher clock period than the other fixed-point models. It is assumed that the increased number of LUTs used to implement the proposed approximate MAC cause additional path delays during routing. When comparing the two MAC models, the rounded MAC model has a 10% longer clock period than the carry MAC model because the rounding operation is more complex than the carry injection. Regarding the final latency, the rounded MAC model achieves a 49% reduction, and the carry MAC model achieves a 54% reduction compared to the floating-point model. Next, we examine resource usage. Because our proposed approximate MAC units are implemented using logic resources, the utilization of configurable logic blocks (CLBs) and LUT is increased. However, the use of DSP blocks is reduced by approximately 78% compared to the fixed-point models. The memory size is reduced by 66% compared to the floating-point model. When comparing the rounded MAC and carry MAC models, the rounded MAC model uses more CLBs and LUTs, but uses slightly fewer flip-flops (FFs) and shift-register LUTs (SRLs).

A comparison with previous works is shown in Table 6. Our proposed accelerator models can operate at a moderately high 100 MHz frequency. The data size of the memory is an 8 bit fixed point, but owing to the removal of unused bits on the integer part, our proposed models can more efficiently handle data transmission. When compared to work [28], our models use 78% fewer DSP blocks, 30% fewer LUTs, 33% fewer FFs, and half the memory. It is notable that our proposed models have low throughput (GOPs) compared to state-of-the-art accelerators. This is because our proposed models do not have novel optimizations on the loop algorithm and memory access, resulting in high latency, due to memory bottleneck. It can be expected, however, that with the same optimizations applied, our proposed models would have similar performance. Furthermore, optimizing the activation layers would result in much better performance, as our current models rely on DSP operations for activation.

Table 6. Results comparison with previous works.

Model	**[24]**	**[27]**	**[28]**	**This Work (Rounded MAC)**	**This Work (Carry MAC)**
Year	2018	2020	2020	2021	2021
FPGA	Zynq VC7VX485T	Zynq XCZU9EG	Artix XC7A20	Zynq XCZU9EG	Zynq XCZU9EG
Clock (MHz)	-	150	50	100	100
Precision (bit)	16-bit fixed	16-bit floating	8-bit fixed	12/8-bit fixed	12/8-bit fixed
Power (W)	0.676	-	14.13	1.673	1.598
GOPs	20.3	28.8	164.1	0.141	0.141
DSP	406	204	571	123	123
LUT	75,221	25,276	88,756	61,713	57,657
FF	38,577	66,569	42,038	27,863	28,311
BRAM	101	55	218	102	102

5. Conclusions

In this study, we designed a CNN accelerator, using an approximate MAC operator based on a fixed-point data type. The implemented network architecture is the LeNet-5 CNN, which performs handwritten digit classification on the MNIST handwritten digit dataset. The proposed MAC operators are the rounded MAC and carry MAC operators. To perform approximate operations in the adder stage, rounded MAC uses a round-to-zero function, and carry MAC uses carry bits to reduce errors following least-significant-bit truncation. The proposed approximate MAC units are implemented using logic resources on FPGAs instead of high-precision DSP blocks, which are underutilized by low-precision data.

The results revealed that, compared to the floating-point model, our two proposed accelerator models, namely the rounded MAC model and carry MAC model, have 66% reduced memory size, and 46% and 54% reduced latency, respectively. Compared to general fixed-point models, the proposed models use additional CLBs and LUTs to implement the approximate MAC operators, but use approximately 78% fewer DSP blocks.

Author Contributions: Conceptualization, M.C. and Y.K.; methodology, M.C.; software, M.C.; validation, M.C. and Y.K.; investigation, M.C.; resources, M.C. and Y.K.; writing—original draft preparation, M.C.; writing—review and editing, Y.K.; supervision, Y.K.; project administration, Y.K.; funding acquisition, Y.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by MIST under grant number NRF-2019M3F3A1A02072093 and by Ministry of Education under grand number NRF-2020R1F1A1055251.

Acknowledgments: This work was supported by the NRF of Korea funded by the MSIT under Grant NRF-2019M3F3A1A02072093 (Intelligent Semiconductor Technology Development Program). This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) and funded by the Ministry of Education (NRF-2020R1F1A1055251). The EDA tool was supported by IC Design Education Center (IDEC), Korea.

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

References

1. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
2. Li, H.; Lin, Z.; Shen, X.; Brandt, J.; Hua, G. A convolutional neural network cascade for face detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015; pp. 5325–5334.
3. Wu, J.; Leng, C.; Wang, Y.; Hu, Q.; Cheng, J. Quantized convolutional neural networks for mobile devices. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 4820–4828.
4. Yamashita, R.; Nishio, M.; Do, R.K.G.; Togashi, K. Convolutional neural networks: An overview and application in radiology. *Insights Imaging* **2018**, *9*, 611–629. [CrossRef] [PubMed]
5. Xiao, T.; Xu, Y.; Yang, K.; Zhang, J.; Peng, Y.; Zhang, Z. The application of two-level attention models in deep convolutional neural network for fine-grained image classification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015.
6. Vinayakumar, R.; Soman, K.P.; Poornachandran, P. Applying Convolutional Neural Network for Network Intrusion. In Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 13–16 September 2017; pp. 1222–1228.
7. Alzubaidi, L.; Zhang, J.; Humaidi, A.J.; Al-Dujaili, A.; Duan, Y.; Al-Shamma, O.; Santamaría, J.; Fadhel, M.A.; Al-Amidie, M.; Farhan, L. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *J. Big Data* **2021**, *8*, 1–74. [CrossRef] [PubMed]
8. Guo, Z.; Huang, Y.; Hu, X.; Wei, H.; Zhao, B. A Survey on Deep Learning Based Approaches for Scene Understanding in Autonomous Driving. *Electronics* **2021**, *10*, 471. [CrossRef]
9. Anwar, S.; Hwang, K.; Sung, W. Fixed point optimization of deep convolutional neural networks for object recognition. In Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), South Brisbane, QLD, Australia, 19–25 April 2015; pp. 1131–1135.
10. Zhou, S.; Wu, Y.; Ni, Z.; Zhou, X.; Wen, H.; Zou, Y. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth. *arXiv* **2016**, arXiv:1606.06160.
11. Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* **2017**, *18*, 6869–6898.

12. Courbariaux, M.; Bengio, Y.; David, J.P. Training deep neural networks with low precision multiplications. *arXiv* **2014**, arXiv:1412.7024.
13. Gysel, P.; Motamedi, M.; Ghiasi, S. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv* **2016**, arXiv:1604.03168.
14. Zhuang, B.; Shen, C.; Tan, M.; Liu, L.; Reid, I. Towards effective low-bitwidth convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 7920–7928.
15. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.
16. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 26–35.
17. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 45–54.
18. Zhou, Y.; Jiang, J. An FPGA-based accelerator implementation for deep convolutional neural networks. In Proceedings of the 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, China, 19–20 December 2015; Volume 1, pp. 829–832.
19. Ghaffari, S.; Sharifian, S. FPGA-based convolutional neural network accelerator design using high level synthesize. In Proceedings of the 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), Tehran, Iran, 14–15 December 2016; pp. 1–6.
20. Gschwend, D. Zynqnet: An Fpga-Accelerated Embedded Convolutional Neural Network. *arXiv* **2020**, arXiv:2005.06892.
21. Abdelouahab, K.; Bourrasset, C.; Pelcat, M.; Berry, F.; Quinton, J.C.; Serot, J. A Holistic Approach for Optimizing DSP Block Utilization of a CNN implementation on FPGA. In Proceedings of the 10th International Conference on Distributed Smart Camera, Paris, France, 12–15 September 2016; pp. 69–75.
22. Lee, S.; Kim, D.; Nguyen, D.; Lee, J. Double MAC on a DSP: Boosting the performance of convolutional neural networks on FPGAs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2019**, *38*, 888–897. [CrossRef]
23. Wang, D.; Xu, K.; Guo, J.; Ghiasi, S. DSP-efficient hardware acceleration of convolutional neural network inference on FPGAs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 4867–4880. [CrossRef]
24. Chen, W.; Wu, H.; Wei, S.; He, A.; Chen, H. An asynchronous energy-efficient CNN accelerator with reconfigurable architecture. In Proceedings of the IEEE Asian Solid-State Circuits Conference (A-SSCC), Tainan, Taiwan, 5–7 November 2018; pp. 51–54.
25. Giardino, D.; Matta, M.; Silvestri, F.; Spanò, S.; Trobiani, V. FPGA implementation of hand-written number recognition based on CNN. *Int. J. Adv. Sci. Eng. Inf. Technol.* **2019**, *9*, 167–171. [CrossRef]
26. Rongshi, D.; Yongming, T. Accelerator implementation of Lenet-5 convolution neural network based on FPGA with HLS. In Proceedings of the 2019 3rd International Conference on Circuits, System and Simulation (ICCSS), Nanjing, China, 13–15 June 2019; pp. 64–67.
27. Shi, Y.; Gan, T.; Jiang, S. Design of Parallel Acceleration Method of Convolutional Neural Network Based on FPGA. In Proceedings of the 2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), Chengdu, China, 10–13 April 2020; pp.133–137.
28. Shan, D.; Cong, G.; Lu, W. A CNN Accelerator on FPGA with a Flexible Structure. In Proceedings of the 2020 5th International Conference on Computational Intelligence and Applications (ICCIA), Beijing, China, 19–21 June 2020; pp. 211–216.
29. Xiao, T.; Tao, M. Research on FPGA Based Convolutional Neural Network Acceleration Method. In Proceedings of the 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), Dalian, China, 28–30 June 2021; pp. 289–292.
30. UltraScale Architecture DSP Slice User Guide. 2020. Available online: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf (accessed on 28 July 2021).
31. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
32. The MNIST Database of Handwritten Digits. Available online: http://yann.lecun.com/exdb/mnist/ (accessed on 29 July 2021).
33. Zeiler, M.D.; Fergus, R. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv* **2013**, arXiv:1301.3557.
34. Yu, D.; Wang, H.; Chen, P.; Wei, Z. Mixed Pooling for Convolutional Neural Networks. In Proceedings of the Rough Sets and Knowledge Technology: 9th International Conference (RSKT 2014), Shanghai, China, 24–26 October 2014; pp. 364–375.
35. Sun, M.; Song, Z.; Jiang, X.; Pan, J.; Pang, Y. Learning pooling for convolutional neural network. *Neurocomputing* **2017**, *224*, 96–104. [CrossRef]
36. Ramachandran, P.; Zoph, B.; Le, Q.V. Searching for activation functions. *arXiv* **2017**, arXiv:1710.05941.
37. Karlik, B.; Olgac, A.V. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *Int. J. Artif. Intell. Expert Syst.* **2011**, *1*, 111–122.
38. Reduce Power and Cost by Converting from Floating Point to Fixed Point. 2017. Available online: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf (accessed on 28 July 2021).